

Implementation of algorithms and data structures

2. 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 6. října 2026

Licence: Creative Commons BY-NC-SA 4.0

Unit testing

Motivation

Verify the correctness of new features and preserve existing functionality after changes to the code.

Description

- Software testing method by which individual units of source code are tested to determine whether they are fit for use
- Tests should be independent

Advantages

- Unit tests can be run repeatedly
- Fast discovery of a bug when code is changed
- Example of the usage of a library

Limitations and disadvantages

- Unit tests only prove that a program contains a bug
- Unit tests cannot prove that a program is correct (halting problem)
- Unit tests are not supposed to verify integration of modules
- Unit tests use the API without verifying the correctness of internal data

Unit testing: Sum of two numbers

```
1 public class TestAdder {
2     @Test
3     public void testSumPositiveNumbersOneAndOne() {
4         Adder adder = new AdderImpl();
5         TEST(adder.add(1, 1) == 2);
6     }
7     @Test
8     public void testSumPositiveNumbersOneAndTwo() {
9         Adder adder = new AdderImpl();
10        TEST(adder.add(1, 2) == 3);
11    }
12    @Test
13    public void testSumPositiveNumbersTwoAndTwo() {
14        Adder adder = new AdderImpl();
15        TEST(adder.add(2, 2) == 4);
16    }
17    @Test
18    public void testSumZeroNeutral() {
19        Adder adder = new AdderImpl();
20        TEST(adder.add(0, 0) == 0);
21    }
22    ...
23 }
```

Unit testing: Graph

```
1 public abstract class AbstractGraphTest {
2     MutableGraph<Integer> graph;
3     @Before
4     public void init() {
5         graph = createGraph();
6     }
7     @After
8     public void validateGraphState() {
9         validateGraph(graph);
10    }
11    @Test
12    public void nodes_oneNode() {
13        addNode(N1);
14        assertThat(graph.nodes()).containsExactly(N1);
15    }
16    @Test
17    public void nodes_noNodes() {
18        assertThat(graph.nodes().isEmpty());
19    }
20    @Test
21    public void adjacentNodes_oneEdge() {
22        putEdge(N1, N2);
23        assertThat(graph.adjacentNodes(N1)).containsExactly(N2);
24        assertThat(graph.adjacentNodes(N2)).containsExactly(N1);
25    }
26 }
```

Source: Wikipedia: Unit testing

<https://github.com/google/guava/blob/master/guava-tests/test/com/google/common/graph/AbstractGraphTest.java>

Initial questions about unit tests

- How should a unit be tested?
E.g., which library should be used to write tests?
- What should be tested?
E.g., what should be the content of unit tests?

Libraries for unit tests

- Python: unittest — Unit testing framework
- Julia: Unit testing (standard library)
- Java: JUnit 5
- C#: Unit test basics
- C/C++: Tens of libraries

Wikipedia: List of unit testing frameworks

What should be tested?

General hints

- Write a test verifying that the task is fulfilled
- Test boundary cases

Test sorting function

```
1 // First, small simple tests
2 TEST(sort([5,7,9,3]) == [3,5,7,9])
3 TEST(sort(['d','a','z']) == ['a','d','z'])
4 TEST(sort(["one","two","three"]) == ["one","three","two"])
5
6 // Tricky cases, some cases may depend on documentation
7 TEST(sort([]) == [])
8 TEST(sort([1,2,1,2,1]) == [1,1,1,2,2])
9 TEST(sort([1,False,5,True]) == [False, 1, True, 5])
```

Creating larger and random tests will be discussed later.

Examples of tests for binary search trees

- 1 Create a new empty tree and destroy it
- 2 Create a new empty tree, insert one element and destroy the tree
- 3 Insert more elements
- 4 Delete some elements
- 5 Delete all elements
- 6 Combine insertion and deletions
- 7 Check the element counter
- 8 Find existing and non-existing elements
- 9 Insert existing and delete non-existing elements
- 10 Find and delete an element in an empty tree

Testing representation invariants

Disadvantages of unit tests

- In non-trivial situations, all cases cannot be tested
- When a unit test fails, it does not say where the bug is
- Unit tests do not verify the correctness of stored data
- E.g., insertion of an element may be incorrect, but a bug may occur when it is deleted

Representation invariant check

A function which traverses the whole data structure and, in one pass, verifies all invariants and all conditions given by its definition.

What can be tested in stored data?

- All conditions given in a definition of a data structure
- Invariants
- Properties implied by proofs of correctness of an algorithm
- Values of variables which can be computed from other data
e.g., the number of elements in a tree
- Values of all variables have expected values
e.g., the range of integers, enumerators

Example: Doubly linked list

```
1 void check_rep_list(list *l) {  
2     if(!l->first) { // List is empty  
3         TEST(!l->last);  
4         return;  
5     }  
6     TEST(l->last);  
7     TEST(!l->first->prev);  
8     TEST(!l->last->next);  
9     for(node *n = l->first; n; n = n->next) {  
10        if(n != l->first)  
11            TEST(n->prev && n->prev->next == n);  
12        if(n != l->last)  
13            TEST(n->next && n->next->prev == n);  
14    }  
15 }
```

Do we check everything?

Example: Doubly linked list (improved check)

```
1 void check_rep_list(list *l) {
2     if(!l->first) {
3         TEST(!l->last);
4         TEST(l->size == 0);
5         return;
6     }
7     TEST(l->last);
8     TEST(!l->first->prev);
9     TEST(!l->last->next);
10    int size = 0;
11    for(node *n = l->first; n && size <= l->size; n = n->next) {
12        ++size;
13        if(n != l->first)
14            TEST(n->prev && n->prev->next == n);
15        if(n != l->last)
16            TEST(n->next && n->next->prev == n);
17    }
18    TEST(size == l->size); // The list ends after l->size nodes
19 }
```

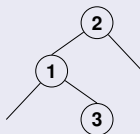
①

Example: binary search tree

```
1 void check_rep_tree(tree *tree) { check_rep_node(tree->root); }
2
3 void check_rep_node(node *node) {
4     if(!node)
5         return;
6
7     TEST(!node->left || node->left->parent == node);
8     TEST(!node->right || node->right->parent == node);
9
10    TEST(!node->left || node->left->key <= node->key);
11    TEST(!node->right || node->right->key >= node->key);
12
13    check_rep_node(node->left);
14    check_rep_node(node->right);
15 }
```

Question

Does this test guarantee that a binary search tree satisfying it is correct?



②

- 1 Note that we cannot test whether there are unreachable nodes. We will design another test checking missing data.
- 2 The check passes for the tree in the figure, yet the tree is not a correct binary search tree. The root 2 has a left child 1, and 1 has a right child 3. Both parent-child inequalities hold locally ($1 \leq 2$ and $3 \geq 1$), but 3 lies in the left subtree of 2 and $3 > 2$, so the required order is violated. The parent-child inequalities do not imply the global binary search tree property.

Testing order in a binary search tree

```
1 // Returns a pair of the minimal and the maximal key in the subtree
2 pair<int,int> check_rep_order(node *node) {
3     int min = node->key, max = node->key, cmp;
4
5     if(node->left) {
6         min,cmp = check_rep_order(node->left);
7         TEST(cmp <= node->key);
8     }
9
10    if(node->right) {
11        cmp,max = check_rep_order(node->right);
12        TEST(node->key <= cmp);
13    }
14
15    return pair(min, max);
16 }
```

Example: Priority queue in an array

```
1 void check_rep_heap(queue *q) {  
2     TEST(0 <= q->size && q->size <= q->capacity);  
3  
4     for(int i = 1; i < q->size; i++)  
5         // Parent is stored on position floor((i-1)/2)  
6         TEST(q->array[i].priority >= q->array[(i-1)/2].priority);  
7 }
```

Hash table with separate chains

- A linked list in every bucket is correct
- Compute the hash of every element to test whether it is stored in the proper bucket
- The ratio of the number of elements to the number of buckets is within the expected range

Graphs

- Incidence lists contain the expected values
e.g., indices of vertices are within the expected range or pointers give vertices inside the graph
- Every edge is a member of incidence lists of both end-vertices

AVL tree

Binary search tree where, for every vertex, the difference between the heights of the left and the right subtree is at most one.

Write a representation invariant check for an AVL tree

- There is a template on GitLab
- Implement one function checking the representation invariants of an AVL tree
- Some unit tests are given, but there are more of them on Recodex

Motivation

- We need to test some parts of the code which are rarely executed
 - E.g., some cases in red-black trees
- It is hard to create a tested configuration using the API
 - E.g., how to find a sequence of operations Insert/Delete for every case in red-black trees?
- Fuzz tests may create a tested configuration, but the data are too large for easy debugging

Manually created data in memory: Basic idea

```
1 class Node:
2     def __init__(self, key, left, right, parent, is_black, size):
3         self.key = key
4         self.left = left
5         self.right = right
6         self.parent = parent
7         self.is_black = is_black
8         self.size = size
9
10 def test_delete_case_uncle_is_black():
11     root = Node(10, None, None, None, True, 6)
12     p = root.left = Node(5, None, None, root, False, 4)
13     l = p.left = Node(2, None, None, p, True, 2)
14     u = root.right = Node(15, None, None, root, True, 1)
15     ...
16     check_rep(root)
17     root.delete(2)
18     check_rep(root)
19     TEST(root.left == u)
20     TEST(u.is_black == False)
21     ...
```

Discussion

- Simple approach to create one small test
- Impractical to create a larger test or multiple tests
 - Setting every variable is time-consuming,
 - hard to read, and
 - leads to unintentional bugs.

Approach

- Encode data into a compact and clear format, e.g.
 - XML, JSON
 - Custom format, e.g. (3,(1,(0,(),()),(2,(),())),(4,(),()))
`https://gitlab.kam.mff.cuni.cz/datovsky/assignments/-/blob/e9a90994c0373599d5140341c567780f226cbf04/02-splay\_operation/cpp/splay\_tests.txt`
 - Use the syntax of the programming language used
- Write a function reading the chosen format

Encode data: example

```
1 def subtree(key, is_black, left=None, right=None):
2     node = Node()
3     node.key = key
4     node.is_black = is_black
5     node.left = left
6     node.right = right
7     node.size = 1 + (left.size if left else 0) +
8                     (right.size if right else 0)
9     if left:
10         left.parent = node
11     if right:
12         right.parent = node
13     return node
14
15 root =
16     subtree(5, True,
17         subtree(3, False,
18             None,
19             subtree(4, True)),
20         subtree(7, True))
```

Goal

Test that the resulting tree after an operation is as expected.

Motivation

Testing every single variable of every node is really tedious, e.g.

```
1 TEST(root.key == 10)
2 TEST(root.left.is_black == False)
3 TEST(root.right.left.left.right.left == None)
4 ...
```

Approach

- Encode input and expected output trees
- Decode both trees
- Run the tested operation on the decoded input
- Compare the result with the decoded expected result

Compare obtained and expected results

```
1 def compare_trees(tested_tree, expected_tree):
2
3     def recursive(tested_node, expected_node):
4         TEST(tested_node.key == expected_node.key)
5         TEST(tested_node.is_black == expected_node.is_black)
6         if expected_node.left:
7             TEST(tested_node.left)
8             recursive(tested_node.left, expected_node.left)
9         else:
10            TEST(not tested_node.left)
11    ...
12
13    recursive(tested_tree.root, expected_tree.root)


1 tested_tree = Tree(...)
2 tested_tree.check_rep()
3 tested_tree.insert(5)
4 tested_tree.check_rep()
5 expected_tree = Tree(...)
6 expected_tree.check_rep()
7 compare_trees(tested_tree, expected_tree)
```

Unit tests

- Test individual units of the code through the API
- Test boundary and tricky cases
- Cannot prove correctness (halting problem)

Representation invariant check

- One function traverses the data structure and verifies all invariants and conditions given by its definition
- Verified after every operation in tests

Manually created data in memory

- Encode compact data (XML, JSON, custom format) and decode it
- Verify the result of an operation against an encoded expected tree

- It is hard to see what is stored in memory
- Without knowledge of the stored data, debugging is difficult
- Visualize stored data!
- The content of an array can be easily printed on the terminal
- Advanced structures need graphical presentation

Write the structure of a red-black tree

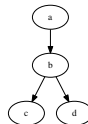
```
1 void rb_print(rb_tree *tree) {
2     rb_print(tree, tree->root);
3     printf("\n");
4 }
5
6 void rb_print(rb_tree *tree, rb_node *node) {
7     if(!node)
8         printf("L");
9     else {
10        printf("(");
11        rb_print(tree, node->left);
12        printf("%d", node->key);
13        if(node->is_red())
14            printf("R");
15        rb_print(tree, node->right);
16        printf(")");
17    }
18 }
```

Output

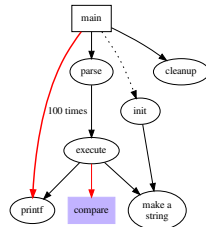
- (((L1L)2(L3L))4(((L6RL)7L)10((L14RL)20L)))
- Output can be improved using ASCII-art
- Graphical visualization may be clearer

Examples of graphs created in DOT

```
1 digraph graphname {  
2   a -> b -> c;  
3   b -> d;  
4 }
```



```
1 digraph G {  
2   size = "4,4";  
3   main [shape=box]; /* this is a comment */  
4   main -> parse [weight = 8];  
5   parse -> execute;  
6   main -> init [style=dotted];  
7   main -> cleanup;  
8   execute -> {make_string printf}  
9   init -> make_string;  
10  edge [color=red]; // so is this  
11  main -> printf [style=bold,label="100 times"];  
12  make_string [label="make a\nstring"];  
13  node [shape=box,style=filled,color=".7 .3 1.0"];  
14  edge [color=red];  
15  execute -> compare;  
16 }
```



More examples: <https://graphviz.org/gallery/>

Write the structure of a red-black tree using DOT

```
1 void rb_print(rb_tree *tree) {
2     printf("digraph G {\n");
3     rb_print(tree, tree->root);
4     printf("}\n");
5 }
6
7 void rb_print(rb_tree *tree, rb_node *node) {
8     if(node) {
9         printf("%d [color=%s];\n", node->key, node->is_red ? "red" :
10             "black");
11         if(node->parent)
12             printf("%d -> %d;\n", node->parent->key, node->key);
13         rb_print(tree, node->left);
14         rb_print(tree, node->right);
15     }
```

Create a Graphviz picture in Python

```
1 import pydot
2
3 graph = pydot.Dot("my_graph", graph_type="graph", bgcolor="yellow")
4
5 # Add nodes
6 my_node = pydot.Node("a", label="Foo")
7 graph.add_node(my_node)
8 # Or, without using an intermediate variable:
9 graph.add_node(pydot.Node("b", shape="circle"))
10
11 # Add edges
12 my_edge = pydot.Edge("a", "b", color="blue")
13 graph.add_edge(my_edge)
14 # Or, without using an intermediate variable:
15 graph.add_edge(pydot.Edge("b", "c", color="blue"))
16
17 # Output image in png format
18 graph.write_png("output.png")
19
20 # Convert to string
21 output_raw_dot = graph.to_string()
22 # Or, save it as a DOT-file:
23 graph.write_raw("output_raw.dot")
```

Language bindings (API)

- Python
- Java
- MATLAB
- WordPress
- LaTeX: TikZ, dot2tex: A Graphviz to LaTeX converter

Visualization and IDE integrations

- Graphviz (dot) language support for Visual Studio Code
- Graphviz Visual Editor: A web application
- Qt Visual Graph Editor (C++)
- More resources about Graphviz

Other graph visualization tools

- NetworkX
- Gephi
- igraph