

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

Summer semestr 2021/22

Last change 2. března 2022

Licence: Creative Commons BY-NC-SA 4.0

Disadvantages of unit tests

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

What can be tested in data representation?

- 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. number of elements in a tree
- Values of all variables have expected values
e.g. range of integers, enumerators

Example: Doubly linked list

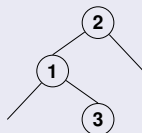
```
1 void list_test(list *l) {
2     if(!l->first) { // List is empty
3         TEST(!l->last);
4         return;
5     }
6     TEST(l->last);
7     TEST(!l->first->prev);
8     TEST(!l->last->next);
9     for(node *n = l->first; n; n = n->next) {
10        if(n != l->first)
11            TEST(n->prev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

Example: binary search tree

```
1 void tree_test(tree *tree) { node_test(tree->root); }
2
3 void node_test(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    node_test(node->left);
14    node_test(node->right);
15 }
```

Question

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



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> order_test(node *node) {
3     int min = node->key, max = node->key, cmp;
4
5     if(node->left) {
6         min,cmp = order_test(node->left);
7         TEST(cmp <= node->key);
8     }
9
10    if(node->right) {
11        cmp,max = order_test(node->right);
12        TEST(node->key <= cmp);
13    }
14
15    return pair(min, max);
16 }
```

Basic property of AVL trees

For every vertex, the difference between heights of the left and the right subtree is at most one.

```
1 // Returns height of node's subtree
2 int height_test(node *node) {
3     if(!node)
4         return 0;
5
6     int left_height = height_test(node->left);
7     int right_height = height_test(node->right);
8
9     TEST(node->height_diff == left_height - right_height);
10    TEST(abs(node->height_diff) <= 1);
11
12    return max(left_height, right_height) + 1;
13 }
```

Example: Priority queue in an array

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

Hash table with separate chains

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

Graphs

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

When these test should be run?

After every atomit operation

During modifications (i.e. insertion, deletion) data may be inconsistent, tests are run when all conditions are expected to be satisfied

Testing during complex operations

Some tests may be run after intermediate steps if we know which conditions should be satisfied, i.e.

- In a heap, its invariant may not be satisfied for at most one node
- In an AVL tree, for at most one node the difference of heights of subtrees is two
- In a red-black tree, at most red node can have red parent

Where calling our testing function should places in code

- Inside unit tests
- Inside functions performing operations
 - Use language tools to turn tests on/off

```
1 #ifdef RUN_CONSISTENCY_TEST
2 tree_test(tree);
3 #endif
```

Testing data representation

- Verifies all invariants on data stored in memory
- Does not test whether some data are not lost or should not be stored
- Needs to be run on some test cases

Which test cases can be used?

- Unit tests with few elements
- Test containing many elements
- Integration tests
- Fuzzy tests

Performance tests are not appropriate since testing consistency may significantly slow

Why should be generate larger tests?

- Tests containing few elements does not check all cases, e.g. in red-black trees
- Manually creating tests with many element is time consuming

Increasing order

```
1 TEST(sorted(list(range(start, stop, step))) == list(range(start, stop, step)))
```

Decreasing order

```
1 TEST(sorted(list(reversed(range(start, stop, step)))) == list(range(start, stop, step)))
```

Random order

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

```
1 array = []
2 for e in range(start, stop, step):
3     array.append(e)
4 for (e,f) in zip(array, range(start, stop, step)):
5     TEST(e == f)
```

```
1 h = Heap()
2 elements = list(range(start, stop, step))
3 random.shuffle(elements)
4 for e in elements:
5     h.push(e)
6 for e in range(start, stop, step):
7     TEST(e == h.pop())
```

Tests containing many elements: Trees

```
1 t = Tree()
2 elements = list(range(start, stop, step))
3 random.shuffle(elements)
4 for e in elements:
5     t.insert(e)
6 for e in elements:
7     TEST(e in t)
8     TEST(not e+1 in t) # assuming step >= 2
9 for (k,e) in enumerate(range(start, stop, step)):
10    TEST(t.find_kth_element(k) == e)
11 random.shuffle(elements)
12 for e in elements:
13    t.remove(e)
14 TEST(t.isempty())
```

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

Write the structure of a red-black tree

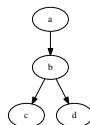
```
1 void rb_print(rb_tree *tree) {
2     rb_print(tree, tree->root);
3     printf("\n");
4 }
5
6 void rb_print(rb_tree *tree, rb_node *node) {
7     if(!node)
8         printf("L");
9     else {
10        printf("(");
11        rb_print(tree, node->left);
12        printf("%d", node->key);
13     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

Output

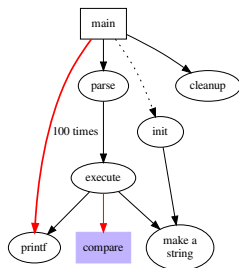
- (((L1L)2(L3L))4((((L6RL)7L)10((L14RL)20L)))
- Output can be improved using ASCII-art
- Graphic visualization may be clearer

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 }
```



Write the structure of a red-black tree using DOT

```
1 void rb_print(rb_tree *tree) {
2     print("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("%s [color=%s];\n", node->key, node->is_red ? "red" :
10             "black");
11         if(node->parent)
12             printf("%s -> %s;\n", node->parent->key, node->parent);
13         rb_print(tree, node->left);
14         rb_print(tree, node->right);
15     }
16 }
```

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

The first assignment: Left-leaning Red-black trees

- Design and implement API
- Design and implement data representation
- Write unit tests with small and also large number of elements
- Write test checking correctness of data representation
- Implement operations insert and find k -th element