Implementation of algorithms and data structures

4. 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 2023/24

Last change 31. října 2023

Licence: Creative Commons BY-NC-SA 4.0

Testing consistency of data representation

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 occur 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

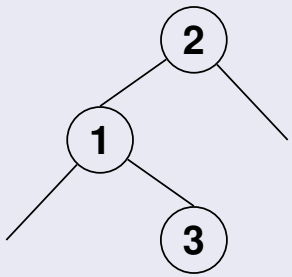
```
1 void list_test(list *l) {
2   if(!l->first) { // List is empty
3     TEST(!l->last);
4     return;
5   }
6   TEST(l->last);
7   TEST(!l->first->prev);
8   TEST(!l->last->next);
9   for(node *n = l->first; n; n = n->next) {
10    if(n != l->first)
11      TEST(n->prev && n->prev->next == n);
12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

Testing height in AVL trees

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 }
```

More examples

Hash table with separate chains

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

Graphs

- Incidence lists contains 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
```

Summary

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 elements is tedious

Tests containing many elements: Sorting

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)))
```

Tests containing many elements: Growing array

```
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)
```

Tests containing many elements: Heap

```
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

```
t = Tree()
elements = list(range(start, stop, step))
random.shuffle(elements)
for e in elements:
  t.insert(e)
for e in elements:
  TEST(e in t)
  TEST(not e+1 in t) # assuming step >= 2
for (k,e) in enumerate(range(start, stop, step)):
  TEST(t.find_kth_element(k) == e)
random.shuffle(elements)
for e in elements:
  t.remove(e)
TEST(t.isempty())
```

Manually created data in memory

Motivation

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

Manually created data in memory: Basic idea

```
class Node:
  def __init__(self, key, left, right, parent, is_black, size):
    self.key = key
    self.left = left
    self.right = right
    self.parent = parent
    self.is_black = is_black
    self.size = size

def test_delete_case_uncle_is_black():
  root = Node(10, None, None, None, True, 6)
  p = root.left = Node(5, None, None, root, False, 4)
  l = p.left = Node(2, None, None, p, True, 2)
  u = root.right = Node(15, None, None, root, True, 1)
  ...
  integrity_test(root)
  root.delete(2)
  integrity_test(root)
  TEST(root.left == u)
  TEST(u.is_black == False)
  ...
```

Manually created data in memory: Basic idea

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 error.

Approach

- Encode data into compact and clear format, e.g.
  - XML, JSON
  - Custom format, e.g. (3,(1,(0,(),()),(2,(),())),(4,(),()))
  - Use syntax of used programming language
- Write a function reading the chosen format

Encode data: example

```
def subtree(key, is_black, left=None, right=None):
  node = Node()
  node.key = key
  node.is_black = is_black
  node.left = left
  node.right = right
  node.size = 1 + (left.size if left else 0) +
                  (right.size if right else 0)
  if left:
       left.parent = self
  if right:
       right.parent = self

root =
  subtree(5, True,
    subtree(3, False,
      None,
      subtree(4, True)),
    subtree(7, True))
```

Manually created data in memory: Other applications

Methods

- Run tested function
- Run data consistency tests on results
- Compare obtained and expected results
- These tests often needs access to private members

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

Visualizate data stored in memory

- 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

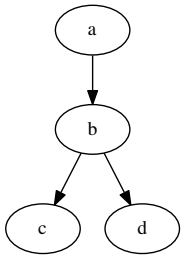
```
1 void rb_print(rb_tree *tree) {
2   rb_print(tree, tree->root);
3   printf("\n");
4 }
5
6 void rb_print(rb_tree *tree, rb_node *node) {
7   if(!node)
8     printf("L");
9   else {
10    printf("(");
11    rb_print(tree, node->left);
12    printf("%d", node->key);
13    if(node->is_red())
14    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

Output

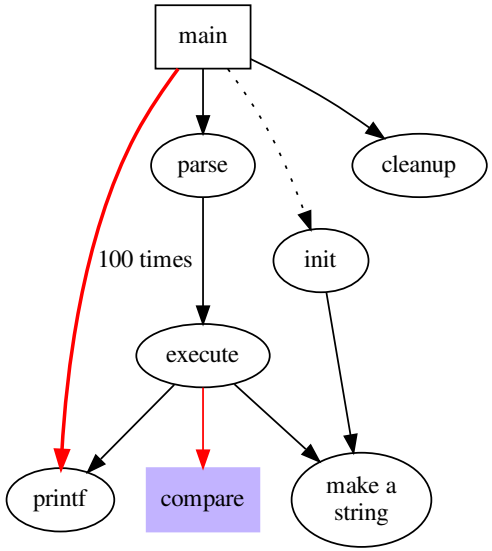
- (((L1L)2(L3L))4(((L6RL)7L)10((L14RL)20L)))
- Output can be improved using ASCII-art
- Graphic visualization may be clearer

Examples of graphs creates in DOT

```
digraph graphname {
   a -> b -> c;
   b -> d;
}
```




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



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" :
          "black");
10    if(node->parent)
11      printf("%s -> %s;\n", node->parent->key, node->key);
12    rb_print(tree, node->left);
13    rb_print(tree, node->right);
14  }
15 }
```

Create a graphviz picture in Python

```
import pydot

graph = pydot.Dot("my_graph", graph_type="graph", bgcolor="yellow")

# Add nodes
my_node = pydot.Node("a", label="Foo")
graph.add_node(my_node)
# Or, without using an intermediate variable:
graph.add_node(pydot.Node("b", shape="circle"))

# Add edges
my_edge = pydot.Edge("a", "b", color="blue")
graph.add_edge(my_edge)
# Or, without using an intermediate variable:
graph.add_edge(pydot.Edge("b", "c", color="blue"))

# Output image in png format
graph.write_png("output.png")

# Convert to string
output_raw_dot = graph.to_string()
# Or, save it as a DOT-file:
graph.write_raw("output_raw.dot")
```

More information about graphviz/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

Tasks for next week

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
- Implement operation delete
- Write fuzz tests
- Debug your program
- Submit your program on recodex