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 16. října 2025

The first assignment

AVL tree

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

Write tests verifying consistency of AVL tree

- There is a template on gitlab
- Implement one function testing consistency of AVL tree
- Some unit tests are given, but there are more of them on recodex

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 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   integrity_test(root)
17   root.delete(2)
18   integrity_test(root)
19   TEST(root.left == u)
20   TEST(u.is_black == False)
21   ...
```

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 unintentional bugs.

Approach

- Encode data into compact and clear format, e.g.
  - XML, JSON
  - Custom format, e.g. (3,(1,(0,(),()),(2,(),())),(4,(),()))
    `https://gitlab.kam.mff.cuni.cz/datovky/assignments/-/blob/e9a90994c0373599d5140341c567780f226cbf04/02-splay_operation/cpp/splay_tests.txt`
  - Use syntax of used programming language
- 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 = self
11  if right:
12       right.parent = self
13  return node
14
15 root =
16  subtree(5, True,
17    subtree(3, False,
18      None,
19      subtree(4, True)),
20    subtree(7, True))
```

Verify result of an operation

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 ...
```

Aproach

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

Compare obtained and expected results

```
def compare_trees(tested_tree, expected_tree):

  def recursive(tested_node, expected_node):
      TEST(tested_node.key == expected_node.key)
      TEST(tested_node.is_black == expected_node.is_black)
      if expected_node.left:
        TEST(tested_node.left)
        recursive(tested_node.left, expected_node.left)
      else:
        TEST(not tested_node.left)
    ...

  recursive(tested_tree.root, expected_tree.root)
```

```
tested_tree = Tree(...)
tested_tree.integrity_tests()
tested_tree.insert(5)
tested_tree.integrity_tests()
expected_tree = Tree(...)
expected_tree.integrity_tests()
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

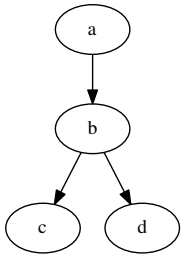
```
1 void rb_print(rb_tree *tree) {
2   rb_print(tree, tree->root);
3   printf("\n");
4 }
5
6 void rb_print(rb_tree *tree, rb_node *node) {
7   if(!node)
8     printf("L");
9   else {
10    printf("(");
11    rb_print(tree, node->left);
12    printf("%d", node->key);
13    if(node->is_red())
14    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

Output

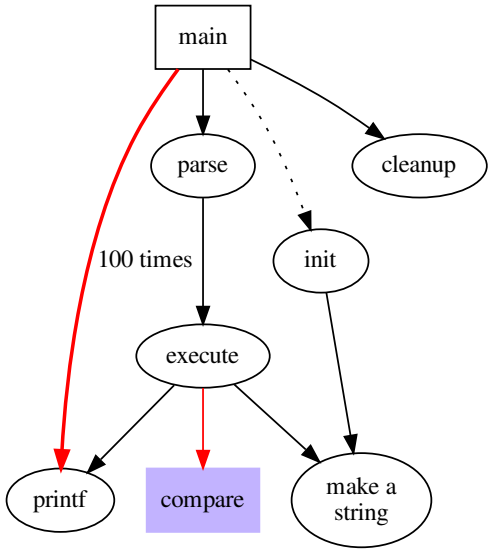
- (((L1L)2(L3L))4(((L6RL)7L)10((L14RL)20L)))
- Output can be improved using ASCII-art
- Graphic visualization may be clearer

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;
}
```



More examples: `https://graphviz.org/gallery/`

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