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

Summer semestr 2021/22

Last change 24. února 2022

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 occurs when it 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

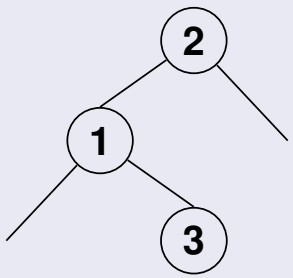
```
1  void list_test(list *l) {
2    if(!l->first) { // List is empty
3      TEST(!l->last);
4      return;
5    }
6    TEST(l->last);
7    TEST(!l->first->prev);
8    TEST(!l->last->next);
9    for(node *n = l->first; n; n = n->next) {
10     if(n != l->first)
11       TEST(n->prev && n->prev->next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

Two issues in data representation of binary trees

- Code symmetry: left and right rotations
- Special case: the empty tree

Splay stromy (Wikipedia)

```
1  template<typename T>
2  class splay_tree {
3    struct node {
4      node *left, *right;
5      node *parent;
6      T key;
7    } *root;
8
9    void left_rotate( node *x ) {
10     node *y = x->right;
11     if(y) {
12       x->right = y->left;
13       if( y->left ) y->left->parent = x;
14       y->parent = x->parent;
15     }
16     if( !x->parent ) root = y;
17     else if( x == x->parent->left ) x->parent->left = y;
18     else x->parent->right = y;
19     if(y) y->left = x;
20     x->parent = y;
21   }
22
23   void right_rotate( node *x ); // Analogous
24 ...
```

Splay stromy (Wikipedia)

```
...
 void splay( node *x ) {
  while( x->parent ) {
   if( !x->parent->parent ) {
    if( x->parent->left == x ) right_rotate( x->parent );
    else left_rotate( x->parent );
   } else if( x->parent->left == x && x->parent->parent->left ==
       x->parent ) {
    right_rotate( x->parent->parent );
    right_rotate( x->parent );
   } else if( x->parent->right == x && x->parent->parent->right ==
       x->parent ) {
    left_rotate( x->parent->parent );
    left_rotate( x->parent );
   } else if( x->parent->left == x && x->parent->parent->right ==
       x->parent ) {
    right_rotate( x->parent );
    left_rotate( x->parent );
   } else {
    left_rotate( x->parent );
    right_rotate( x->parent );
   }
  }
 }
...
```

Splay trees: Disadvantages of implementation from Wikipedia

Disadvantages

- We have two very similar functions from left and right rotations
- We have to test whether pointers to a parent and a child are NULL
- Having two symmetrical functions and the number of conditions easily leads to a bug

The first improvement

Store two pointers to children in an array

Splay trees: Avoid left-right symetry

```
1  template<typename T>
2  class splay_tree {
3    struct node {
4      node *children[2], *parent;
5        T key;
6
7        node* get_left() {
8          return children[0];
9        }
10
11       int get_order() {
12         return parent && parent->children[0] == this ? 0 : 1;
13       }
14     };
15     node *root;
16
17     void set_child(node *parent, node *child, int order) {
18       if(parent)
19         parent->children[order] = child;
20       else
21         root = child;
22       if(child)
23         child->parent = parent;
24     }
```

Splay trees: Avoid left-right symetry

```
1  // Rotate a given node n with its parent
2  void rotate(node *n) {
3    int order = n->get_child_order();
4    node *p = n->parent;
5
6    set_child(p->parent, n, p->get_child_order());
7    set_child(p, n->children[1-order], order);
8    set_child(n, p, 1-order);
9  }
10
11 void splay(node *n) {
12   while(n != root) {
13     if(n->parent == root) // Zig
14       rotate(n);
15     else if(n->get_child_order() == n->parent->get_child_order()) {
16       rotate(n->parent); // Zig-zig
17       rotate(n);
18     }
19     else { // Zig-zag
20       rotate(n);
21       rotate(n);
22     }
23   }
24 }
```

Splay trees: Discussion

Advantages

- We have one function for both left and right function
- No condition in our rotation
- Small auxiliary functions simplifies code

Next step

- Remove special conditions for empty trees
- Remove conditions for a pointer to the parent being NULL
- Introduce an auxiliary node storing a pointer to the root in the left child

Splay trees: auxiliary node for the root

```
1  template<typename T>
2  class splay_tree {
3    struct node {
4      node *children[2], *parent;
5        T key;
6
7        int get_order() {
8          return parent->children[0] == this ? 0 : 1;
9        }
10   };
11   node head; // Init: head.children[0] = nullptr
12
13   void set_child(node *parent, node *child, int order) {
14     parent->children[order] = child;
15     if(child)
16       child->parent = parent;
17   }
```

Tasks for next week

The first assignment: Left-leaning Red-black trees

- Create a new private git repository on `https://gitlab.mff.cuni.cz/`
- Give me (finkj1am) access (developer)
- Read and understand literature
- Design and implement API
- Design and implement data representation
- Write unit tests
- Write test checking correctness of data representation
- Use git every day!