

1 Routings in transformation program

```
input : program, program to be processed; ast, abstract syntax tree of the program
output: new_program, processed program

1 new_program  $\leftarrow \emptyset$ 
2 foreach block  $\in$  program do
3   new_block  $\leftarrow \emptyset$ 
4   pointers[ ]  $\leftarrow \emptyset$ 
5   eval_prog  $\leftarrow$  Construct a random evaluation program from the evaluation templates
6   block  $\leftarrow$  Mix_Program(block, eval_prog)
7   constant_pool  $\leftarrow$  [All constants in the block]
8   CT  $\leftarrow$  build_constant_trees(constant_pool, pointers[ ])
9   CT_val  $\leftarrow$  decode(CT)
10  append(new_block, dynamic_build_constant_tree(CT_val))
11  foreach statement  $\in$  block do
12    if have_constant(statement, ast) then
13      foreach constant  $\in$  statement do
14        pointer  $\leftarrow$  locate_subtree(pointers[ ], constant)
15        substitute(statement, constant, decode(pointer))
16      end
17    end
18    val  $\leftarrow$  left value of the statement
19    append(new_block, statement)
20    append(new_block, update_graph(CT, val))
21  end
22  append(new_program, new_block)
23 end
24 return new_program
```

Algorithm 1: Overview

```
input : constant, constant to be search in the tree
        pointers[ ], a set of pointers used to point to the
        root of the subgraphs representing the constant
output: pointer, the root of the subgraph representing the constant
        NULL, if not find

1 if constant  $\in$  pointers[ ] then
2   return pointer
3 else
4   return NULL
5 end
```

Algorithm 2: Locate subtree

input : constant_pool, a set of constants used in the block;
 pointers[], a set of pointers used to point to the
 root of the subgraphs representing the constant
output: CT, the root of the constant tree

```

1 CT ← Root of init_tree
2 foreach constant ∈ constant_pool do
3   subtree_root ← build_const_tree(constant)
4   node ← locate_subtree(pointers[ ], constant)
5   if node == NULL then //if cannot find one, construct one
6     node ← random_node(CT)
7     merge_tree(node, subtree_root)
8   else //if find one, make a duplication in probability
9     toss a coin
10    if head is up then
11      node ← random_node(CT)
12      merge_tree(node, subtree_root)
13    end
14  end
15  pointers[ ].join((node, constant ))
16 end
17 return CT

```

Algorithm 3: Build Constant Trees

2 Routings in watermarked program

input : CT, the root of the constant tree; val, value update refers to
output: None

```

1 node ← find_upgradable_node(CT)
2 if node == node →parent →left then
3   parent_point ← &node →parent →left
4 else
5   parent_point ← &node →parent →right
6 end
7 if val is odd then //left rotate
8   switch rotate case do
9     case 1 left_rotate_1()
10    case 2 left_rotate_2()
11    case 3 left_rotate_3()
12    case 4 left_rotate_4()
13  end
14 else
15   switch rotate case do
16     case 1 right_rotate_1()
17     case 2 right_rotate_2()
18     case 3 right_rotate_3()
19     case 4 right_rotate_4()
20   end
21 end
22 if CT →left ≠ CT →right then //update the child pointers of the root
23   CT →right ← CT →left
24 end

```

Algorithm 4: UpdateGraph

```

*parent_point ← node →right;
node →right →parent ← node →parent;
node →parent ← node →left;
node →right ← node →parent →left;
node →parent →left →parent ← node;
node →parent →left ← node;

```

Algorithm 5: Left rotate case 1

```

allocate(new_node);
new_node →left ← node →right →left;
new_node →right ← node →right →right;
node →right →left →parent ← new_node;
node →right →right →parent ← new_node;
node →right →right ← new_node;
node →right →left ← node;
node →right →parent ← node →parent;
*parent_point ← node →right;
node →parent ← node →right;
allocate(another_new_node);
node →right ← another_new_node;
pointers[ ].update(node →parent, node →parent →right);

```

Algorithm 6: Left rotate case 2

```

*parent_point ← node →left;
node →left →parent ← node →parent;
node →parent ← node →right;
node →left ← node →parent →right;
node →parent →right →parent ← node;
node →parent →right ← node;

```

Algorithm 7: Right rotate case 1

```

allocate(new_node);
new_node →left ← node →right →left;
new_node →right ← node →right →right;
node →right →left →parent ← new_node;
node →right →right →parent ← new_node;
node →right →right ← new_node;
node →right →left ← node;
node →right →parent ← node →parent;
*parent_point ← node →right;
node →parent ← node →right;
allocate(another_new_node);
node →right ← another_new_node;
pointers[ ].update(node →parent, node →parent →right);

```

Algorithm 8: Right rotate case 2

3 Routings shared in both parts

Algorithm 9: Build Constant Tree

Algorithm 10: decode

Right Rotate(Case1)

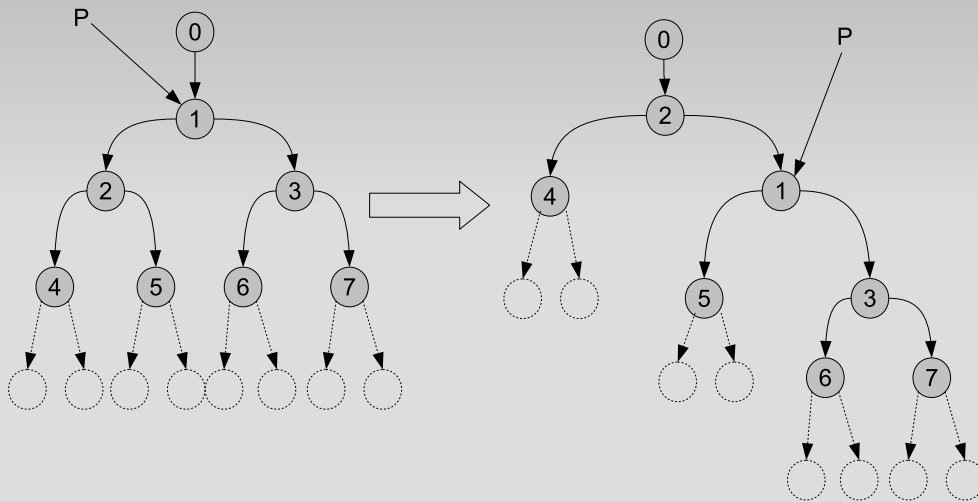


Figure 1: Right Rotate Case 1

Right Rotate(Case2)

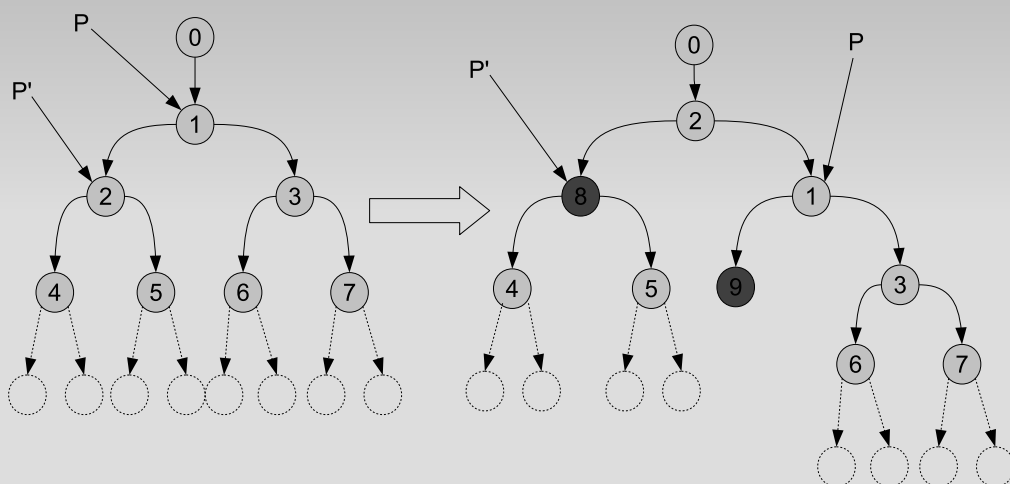


Figure 2: Right Rotate Case 2

Right Rotate(Case3)

The diagram illustrates a Right Rotate operation in Case 3. It shows two binary trees connected by a large arrow, indicating a transformation.

Left Tree (Before Rotation):

- Root: 0
- Node 0's left child: 1
- Node 1's left child: 2
- Node 1's right child: 3
- Node 3's left child: 4
- Node 3's right child: 5
- Nodes 4 and 5 have dashed arrows pointing to empty circles, indicating they are leaf nodes.
- Node 1 is pointed to by a pointer labeled 'P'.

Right Tree (After Rotation):

- Root: 0
- Node 0's left child: 2
- Node 2's left child: 6 (shaded dark gray)
- Node 2's right child: 1
- Node 1's left child: 7 (shaded dark gray)
- Node 1's right child: 3
- Node 3's left child: 4
- Node 3's right child: 5
- Nodes 4 and 5 have dashed arrows pointing to empty circles, indicating they are leaf nodes.
- Node 1 is pointed to by a pointer labeled 'P'.

Figure 3: Right Rotate Case 3

Right Rotate(Case4)

The diagram illustrates a Right Rotate operation (Case 4) on a B-tree. The initial state shows a pointer P pointing to a leaf node containing the value 1, which is the right child of an internal node containing the value 0. The final state shows the result of a right rotation: the internal node now contains the value 2, its left child is a leaf node containing the value 3, and its right child is a leaf node containing the value 1. The pointer P now points to the leaf node containing the value 1.

Figure 4: Right Rotate Case 4