Symmetry brought back to red-black tree top-down deletion

A preliminary report

Luděk Kučera
Algovision.org
and
Faculty of Mathematics and Physics
Charles University
Prague
and
Faculty of Information Technologies
Czech Technical University
Prague

September 16, 2021

1 Introduction

Red-Black (RB) trees are with us already more than four decades, [2, 4]. They are binary search trees modified so that their depth is kept below $2 \log_2 n$. This is why all operations that do not modify the shape of a tree are identical with simple BST operations.

Two standard operations that *do* modify the shape of a red-black tree are insertion and deletion. As usual, the deletion is more difficult. There are several implementations of a RB deletion. As a standard reference, one can cite the algorithm described in [3].

R. Sedgewick [5] proposed another approach to RB insertion and deletion that is defined for a restricted class of RB-trees, so called Left-leaning Red-

Black (LLRB) trees, where no node with a right child can be without a left child. The goal was to develop algorithms that are very simple and have an extremely short code that is free of mirror copies of functions.

While this goal was successfully reached, a disadvantage of LLRB trees is their lack of symmetry. E.g., the deletion code in [5] has quite different parts to process the left and the right branch of a subtree of the actual node. While the left part is simple (and hence deleting the minimum is presented as a starting example, because it moves always left), the right part is more complex and not too much intuitive.

LLRB algorithms are formulated as top-down process that starts in the root and operates along the path from the root to the node to be deleted. It is argued in [5] that this is an advantage, because bottom-up deletion (e.g., [3]) requires parent pointers that are wasting memory.

Bottom-up methods do not need *all* parent pointers; it would be sufficient to build a path from the root to the node of interest is found first and then to follow it backwards.

However, the algorithms of LLRB are not just top-down, but top-down *plus* bottom-up. They involve a *fix-up* phase that, after inserting or deleting of a node, passes backwards along the search path and cleans the tree (eliminates possible edges connecting two red nodes, right-leaning nodes and, in the case of 2-3 trees, black nodes with two red children).

So both classes (CLRS and LLRB) could be denoted as "top-down-up"; the main difference is that LLRB algorithms do the principal work in the top-down phase, while the bottom-up phase is just cleaning, the top-down phase of the above mention adaptation of bottom-up algorithms is trivial. The usual terminology is related to the working phase of the method.

It is sometimes argued that bottom-up methods are usually more efficient, because modifications of the tree that top-down methods do in their first phase are often unnecessary for a required insertion or deletion; they make changes to prepare for the worst case that eventually need not occur, but the present paper is a part of effort to find advantageous top-down methods.

In the same way as Theseus would not get out of Labyrinth without help of Ariadne, top-down algorithms need to keep track of the search path for the fix-up return. A smart implementation is used in [5]: both insert and delete are implemented as recursive functions, and the Ariadne's thread is kept in the stack of recursion calls:

```
Node process(Node node) {
preProcess(node);
node child = a left or right child of node;
process(child);
fixUp(node);
}
```

While this approach gives an elegant and concise code, a recursion call is a procedure that activates the run-time system software and is out of control of a user and, when compared with nanosecond times for simple assignments and conditional branches, might cause a substantial slow-down of the computation.

2 An informal description of the algorithm

The present paper brings a variant of a top-down deletion algorithm that exhibits the following advantages:

- the algorithm can be applied to an *arbitrary* red-black tree (unlike in the case of LLRB-trees, the class of general RB-trees is closed under a left-right mirror);
- the algorithm is very simple logically and can be implemented by a fairly short code;
- a computation using the algorithm is a *pure top-down process*, no fix-up is needed if only RB-tree conditions are imposed on the resulting tree;
- as a consequence, the algorithm can easily be implemented as a non-recursive iterative loop;
- it is conceivable that the algorithm might be generalized to a wider class of trees than RB-trees (to trees that would correspond to B-trees with wider nodes).

It is well known that deleting a node of a BST that has both children can easily be reduced to deleting a node with at most one child. Moreover, a node of a RB-tree with exactly one child must be a black node and its only child must be a red leaf. Deletion of such a node is fairly simple. Finally a deletion of a red leaf of a RB-tree is trivial.

Therefore the algorithm as presented here is limited to deleting a leaf of a RB-tree in order to make the presentation simpler. However, in the case of deleting a node with two children, if the activity the algorithm executes along the search path to the node to be deleted is continued along the path to the node that has the largest among smaller keys, a general deletion algorithm can be obtained that shares properties of the leaf-deleting algorithm.

Since a read leaf deletion is trivial, the way our algorithm deletes a black leaf is to transform the tree so that the leaf to be deleted becomes red, and then the leaf is simply deleted.

3 Clusters

If edges connecting black nodes with their parents are removed, a tree is decomposed into components that will be called *clusters*. Each cluster is a tree with a black root, all other nodes are red. Clusters correspond to nodes of a B-tree that is associated with a red-black tree.

We will say that a cluster is *delete-ready* iff it contains at least one red node.

It is obvious that a leaf node in a delete-ready cluster is red - the only black node in a cluster is its root, which, however, has a red child and hence it is not a leaf even with respect to its cluster. (An exception is a cluster containing the root, but if the root is a leaf, the tree has just one node and the deletion is trivial).

Unfortunately, a black leaf represents a one-node cluster that is *not* delete-ready. On the other hand, a red leaf is obviously a member of a cluster that *is* delete-ready. Therefore the goal of our algorithm is to modify, if necessary, the cluster containing the node to be deleted so that the cluster is delete-ready.

The search path of a leaf of a tree is the uniquely determined path from the root of the tree to said leaf.

The algorithm presented here uses a token that is located in the tree root at the beginning, follows the search path, and before it enters a new cluster, the cluster is modified so that it is delete-ready. In this way, when the token arrives to the leaf to be deleted, the leaf is red, being a member of a delete-ready cluster.

A *color flip* in a node that has both children that have the other color than the node is simply a simultaneous change of the colors of the node and

its both children.

If the node is black and has two red children then it is the root of the cluster that involves both its children.

When a color flip is applied to a black node, the node becomes a member of the cluster that contains the node's parent, and the children of the node become roots of two independent clusters. In this case, a color flip causes a cluster split.

Conversely, if a color flip is applied to a red node with two black children, the node is detached from its cluster to become the root of a cluster that has swallowed the clusters originally rooted by the node's children. We speak about a cluster join.

The cluster rooted by the tree root is a partial exception - a color flip will not change the color of the root that is always black. However, even in this case a color flip causes either a split or a join of clusters.

4 A general deletion in a tree with red and black nodes

It is convenient to explain the algorithm in more general setting of trees with red and black nodes that fulfill the black depth condition (any path from the root to a leaf or a node with only one child contains the same number of black nodes - we do not use sentinel nodes) and the black root condition, but can be very far from satisfying the condition of no red edge[1]. We will call such trees RB-like.

We use general RB-like trees, because the underlying idea of the algorithm is not well visible in trees with small clusters like standard RB-trees.

As already said, the algorithm uses a token that follows the search path to the leaf to be deleted (as the tree is modified during the computation, the search path changes as well).

The computation is divided to *iterations*: an iteration is a time interval during which the token stays in one particular cluster. The activity of the algorithm during each particular iteration is divided into three *phases* (each phase can be void, and the last iteration finishes during the first phase):

[1]Some authors, e.g. [5], use the term *a red edge* for an edge connecting a red node with its parent. In the present paper, a "red edge" means an edge that has both nodes (the parent and the child) red. Using this definition, no red edges must exist in any RB-tree.

- **The first phase**: At the beginning of the phase, the token is in the root of a cluster. During the phase, the token follows the search path and stops in the last node of the search path that belong to the same cluster. If, at the end of the first phase, the token node is the leaf to be deleted, the computation finishes by deleting the leaf.

- **The second phase**: While the token node has a red child, the edge from the token node to its red child is rotated. Our implementation of a rotation of an edge that has the token as the upper node is that the token is moved to the other end of the edge which becomes its upper node (we need it in the third phase). However, in this case, the token is immediately moved back to the node in which it has been located before the rotation.

- **The third phase**:

  - If the child of the token node that is on the search path is the root of a delete-ready cluster, the token moves to that child and the third phase and the present iteration is over.
  - Otherwise a color flip is first applied to the token node, which makes the token node to become the root of a joined cluster that involves the original clusters of the token node (originally) black children.
  - If the child of the token node that is not on the search path (a sibling) has been the root of a not-delete-ready (i.e., one node) cluster, the third phase and the present iteration finishes here.
  - If the cluster rooted in the sibling has been delete-ready, then the joined cluster is modified by rotations so that (at the expense of the sibling branch) its search path branch is made bigger. This part of the algorithm will be described later in detail for the case of proper RB-trees.
  - Finally, a color flip is applied to the new root of the joined cluster to split it back to two clusters and to return its root to the parent cluster, and then the token is moved to the root of the cluster that is on the search path.

Let us comment the behavior of the algorithm during one iteration (for an example with figures, see the next section):

At the end of the first phase (assuming that the token node is not the leaf to be deleted), the child of the token node that belongs to the search path is in another cluster. Since the search path enters each cluster in its root, the child is black. The black depth condition implies that the node must have both children. However, the other child of the token node can be red (and, consequently, a member of the same cluster).

If the second phase is not void, each rotation preserves the partition of nodes to clusters, but the root of the active cluster can change if the token was in the cluster root before the rotation.

Each rotation with a move puts the token node one layer down in the tree, and hence the number of rotations is limited. It is obvious that at the end of the second phase the token has two black children, one of them belongs to the search path.

There are three possible executions of the third phase:

- The cluster below the token node and on the search path is delete-ready, and therefore there is no need of any action, just the token moves to its root.
- The children of the token node are both roots of one-node cluster (they have no red children and hence their clusters are not delete-ready). The children clusters are joined together with the token node to form a delete-ready cluster. The joined cluster is too small to be split back so that one part is a delete-ready cluster. The token mode is located in the root of the joined cluster. The phase finishes here.
- The last possibility is that the child cluster on the search path has just one node, the cluster of the other child (a sibling) is bigger. Using a cluster join, rotations of the cluster edges, and a cluster split, some of nodes of the sibling cluster are moved to the cluster on the search path that becomes delete-ready.

5 An example of the general method

The example is a series of figures that are commented in the following text:

Fig. 1: Initial tree: The figure shows a tree with red and black nodes that satisfies the black depth rule, the root is black. However, there are many red edges in the tree (see a terminological footnote above). The goal is to delete

the node labeled 25; the search path is colored green. The token (marked by a small dark green disk), is in the root of the tree.

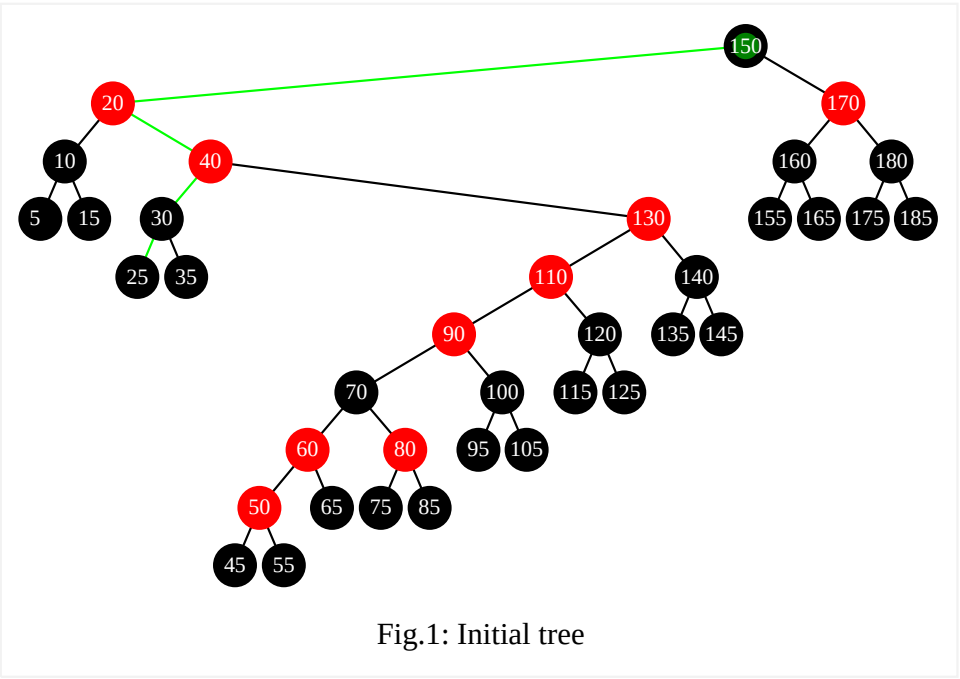

Fig.1: Initial tree

Fig. 2: Initial tree, cluster view: The input tree is shown in another way that makes it easier to see the clusters. It is obvious that during the first phase, the token starts in 150, passes through 20 and finishes in 40, where the search path leaves the cluster.

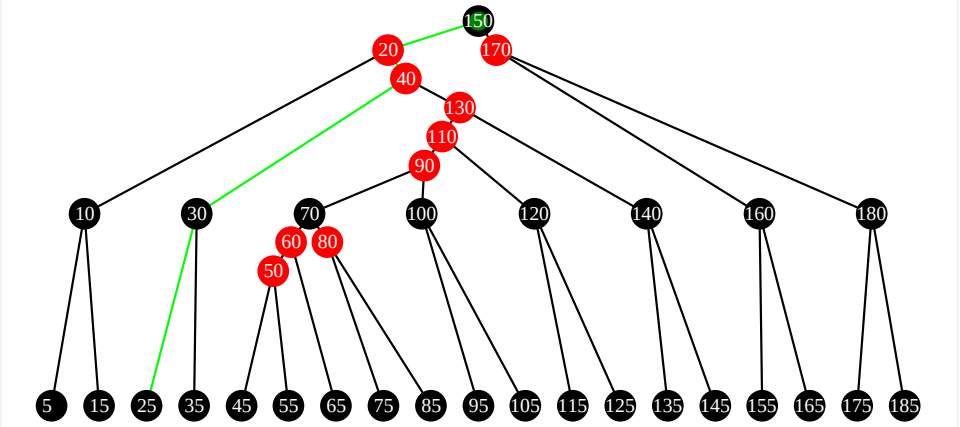

Fig. 2: Initial tree, cluster view, after the 1st phase the token is in 40

Figs. 3,4,5: The tree after the first/second/third rotation of the second phase: After the first phase, the token is in the node 40, which has a red child (not on the search path). Three consecutive rotations of edges to a red child of the token node are performed. The node 40 falls down and after third rotation, it is at the bottom of its cluster and it has two black children: the search path node 30 and the sibling 70.

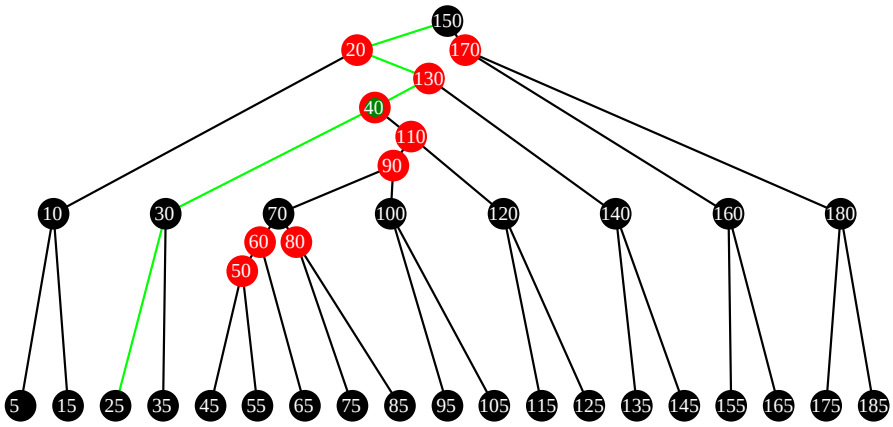

Fig.3: The tree after the first rotation of the second phase

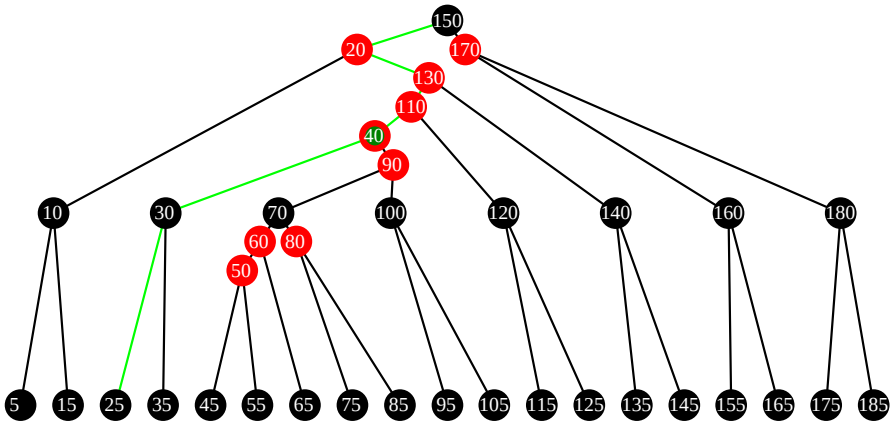

Fig.4: The tree after the second rotation of the second phase

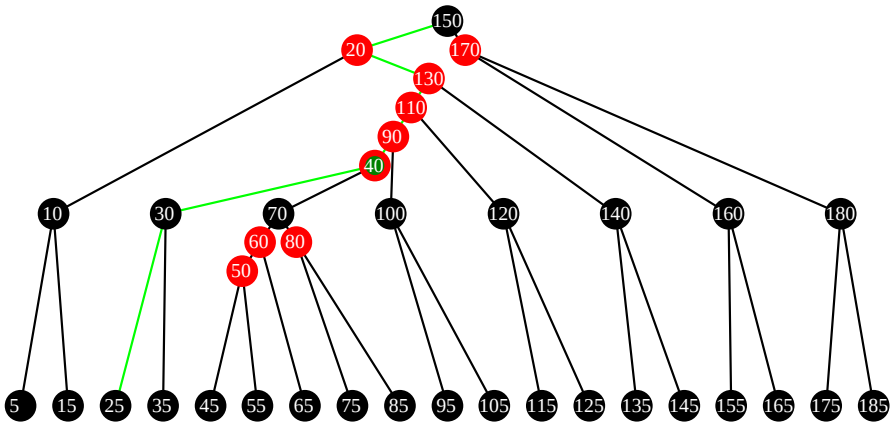

Fig.5: The tree after the third (last) rotation of the second phase

Fig. 6: The tree after the cluster join in the third phase: At the beginning of the third phase, the node 30 forms a one-node cluster that is not delete-ready. A color flip, applied to the token node 40 and its children 30 and 70 creates a big joined cluster, rooted at 40.

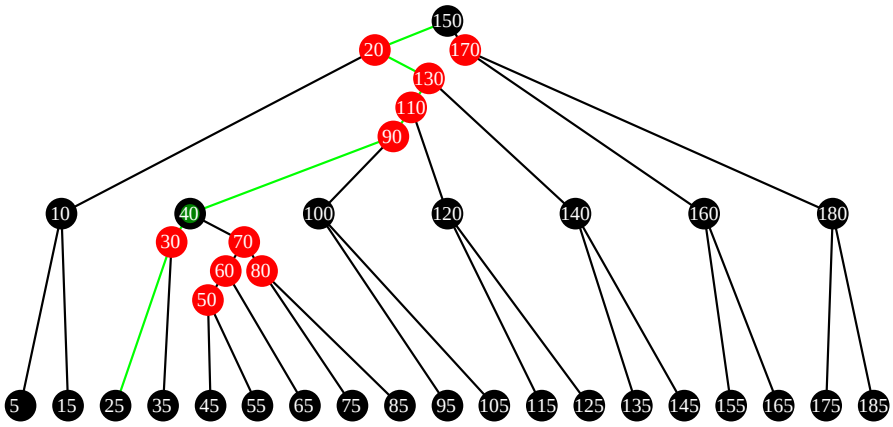

Fig.6: The tree after the cluster join in the third phase

Fig. 7: The tree after the modification of the joined cluster: A single rotation of the edge 40-70 transforms the joined cluster so that its left branch (that is on the search path) is now relatively large. The joined cluster is now rooted by 70. During the rotation, the token moved to the node 70.

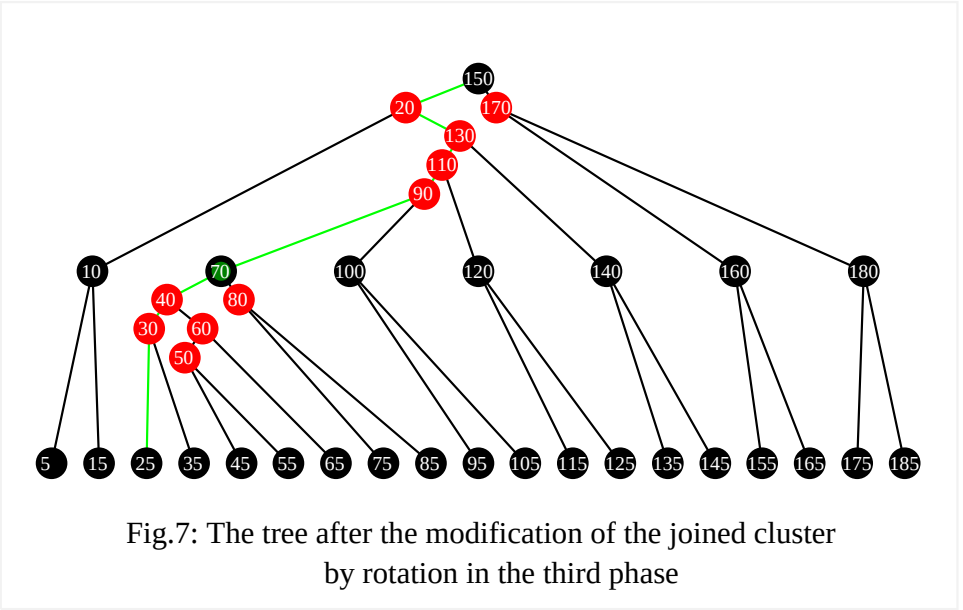

Fig.7: The tree after the modification of the joined cluster
by rotation in the third phase

Fig. 8: The joined cluster split and the token moved to the cluster on the search path: a color flip, applied to the black node 70 and its red children 40 and 80, splits the joined cluster, 70 joins the upper cluster, and 40 and 80 root their own clusters. The token moves from 70 along to the search path to 40 and the third phase as well as the actual iteration is over.

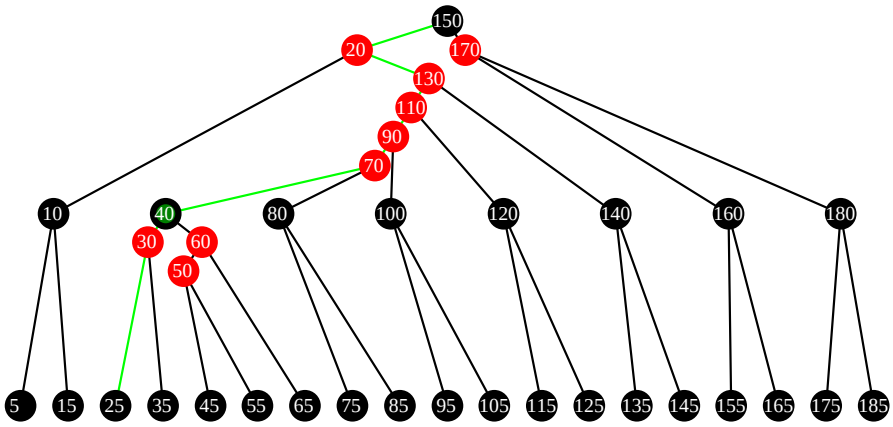

Fig.8 The joined cluster split and the token moved to the cluster on the search path, the third phase finished

6 The special case of standard red-black trees

Now, let us analyze the behavior of the general algorithm in the case of absence of red edges (see a footnote definition of the term given above).

The first phase of any iteration has two possible forms. The edge of the search path from the root of the cluster finishes

- either in a black node, see Fig. 9, the left tree, where edges of the search path are green, and the first phase is void, the token is at the same time in the last search path node of the cluster;
- or in a red node, see Fig 9, the right tree, and the first phase moves the token to that node. Another token move during the first phase is not possible, because the red token node has no red children.

Fig. 9

The second phase has also two possible forms. At the end of the first phase the token is in a node of the cluster such that the next search path node is in another cluster, which is black, because the cluster root is always an entry point for the search path. The black depth rule implies that, at the moment, the token node has the second child as well; let us call it a sibling.

- If the sibling is black, the second phase is void;
- If the sibling (3) is red, see Fig. 10 (where the token node has no color, because it can be black or red), the second phase begins by a rotation of the edge to the sibling, but the token is immediately moved to the node in which it has been before the rotation. A new child (4) of the moved token node has been a child of the red sibling, and hence it is black and therefore the second phase is always limited to at most one rotation. The token node becomes red, but because the node 2 keeps its original black color and 4 is black as well, *no red edge is created*.

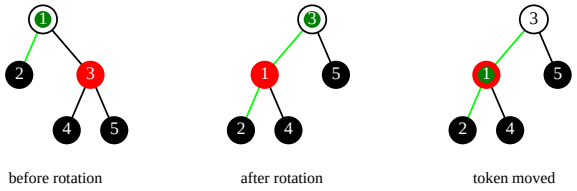

Fig. 10

At the beginning of the third phase the token node has two black children, one of them belongs to the search path and will be called a search child, the other one will be called a sibling.

- If the cluster of the search child is delete-ready, the third phase is simple and consists of moving the token to the search child, which is a root of another cluster. The tree does not change.
- If the search path child has no red child, i.e., its cluster is not delete-ready and has a single node, then the third phase begins by a color flip in the token node, which removes the token node from its cluster to root a joined cluster that swallows the search child cluster and the sibling cluster. The sibling can have 0, 1, or 2 red children (a gray node in the figures means a black child or no child or, equivalently, not a red child):
  - If the sibling has no red children, the third phase finishes, see Fig. 11.

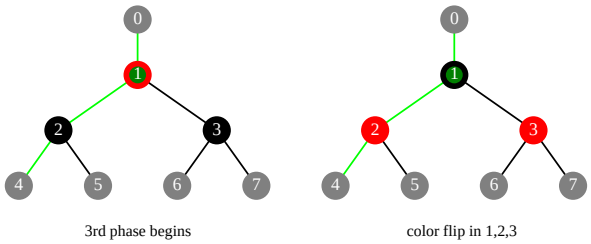

Fig. 11

  - If the sibling is a right child of the token node and has a red right child and no left red child, see Fig. 12, or it is a left child of the

token node and has a red left child and no right red child (a mirror of Fig. 12), or it the sibling has two red children, see Fig. 13, then a single rotation of the edge connecting the token node with the sibling transforms the joined cluster so that its search path branch is not trivial and after the split the search path cluster is delete-ready and a move of the token to its root concludes the third path.

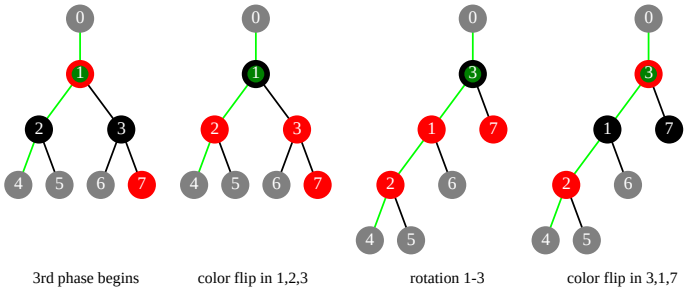

Fig. 12

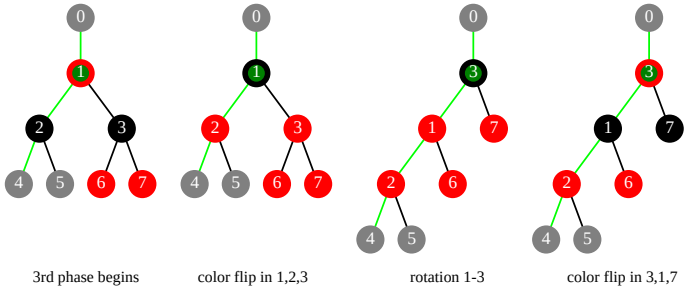

Fig. 13

– if the sibling has just one red child and the sequence the token node : the sibling : the red child of the sibling is a zig-zag, see Fig. 14 or its mirror, then we have to rotate first the edge connecting the sibling with its red child to get the situation from the previous paragraph, and then the third phase is concluded by another rotation, cluster split and a token move.

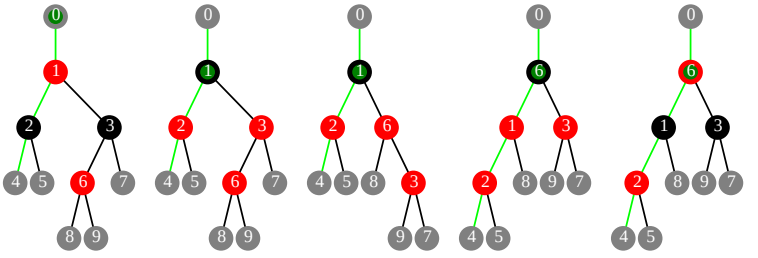

Fig. 14

This shows that, in all cases, the third phase modifies the tree so that the token enters a new cluster that is *delete-ready*. Moreover, again in all cases, *no red edge has been created*.

Altogether, as a result of the previous analysis, one iteration moves the token to the root of a next cluster that has been (if necessary) modified so that it is delete-ready, and the iteration does not create any red edge.

In this way we obtained a single pass top-down iterative algorithm that needs no post-processing (fix-up), because the output tree of an iterative top-down pass is a correct red-black tree.

7 Algovision

An animation of the above general tree example as well as examples of the algorithm applied to RB-trees can convenienly been found is [1].

In the algorithm selection page
select **Data Structures : Red-Black Tree**,
select **Free Work**, and
click **Start Tour**.

When the algoritm tour starts,
use the menu **Scene Choice** to select either **Top Down General Deletion** or **Top Down RB Deletion**, and
animate the computation by clicking the button **Go on**
(or "compute back" by **Back** in the same line).

Clicking **Show Clusters** switches to the cluster view of the tree.

8 A formal description of the algorithm

The code that is presented here is written in C++ and it is not just a publication description, but it is taken from a program that is used to evaluate the properties of the proposed algorithm. It has already executed successfully several millions of delete operations on RB trees of different sizes up to thousands of nodes. The results of tests will be published in the full version of this report.

For a formal description of the algorithm, we can suppose that nodes of trees are objects of the following class `Node` (for simplicity, we assume that node keys are integers; it would be easy to modify the code so that keys are more complex objects that can be compared). Hopefully the names of functions suggest their function (implementation of the headers is given in the Appendix). Several functions have a boolean parameter that selects its side orientation. E.g., child(true) returns the left child of the node, child(false) gives the right child.

The file "node.hpp"

```
class Node {
public:
Node(int key);

Node* leftChild;
Node* rightChild;
bool  isRed;
int   key;

bool  hasRedLeftChild ();
bool  hasRedRightChild ();
bool  hasRedChild (bool lookForLeftChild);
bool  hasRedChildren ();
Node* child(bool leftChildRequested);
void  setChild(bool setLeftChild, Node* child);
}
```

A RB-tree is given as an object of the class `Tree`. While the functions `colorFlip` and `removeLeaf` are obvious (colorFlip does not change the root color when applied to `tree->root`, `removeLeaf` removes a leaf child

of `parent` specified by `removeLeftChild`), some words have to be said about rotation functions.

Usually, parameters of a rotate function specify the upper node of the rotated edge and its orientation (left or right). However, in order to execute a rotation, it is also necessary to know the parent of the given node. This is not a problem when parent pointers are used. The information can also be obtained if a two-pass process is used and rotations are executed during the bottom-up period, when a stack representing the search path in the tree is available either directly or in the form of the recursion call stack of the system.

However, the presented algorithm is a pure top-down process, that does not need to make a search path record. This is why a practical implementation of the algorithm does not use a single token that follows the search path of the leaf to be deleted, but also information about the token's parent. We aggregate this data into an object of the class Token, it is also convenient to know explicitly whether the token is a left or right child of its parent.

The function `Token.move(bool moveLeft)` moves the token to the left or right child of the actual node, depending on the value of the parameter. The function `Token.moveRedIfPossible(bool left)` does the same, provided the target child is red.

The file "token.hpp"

```
#include "node.hpp"
class Token {
public:
Token(Node* node);

Node* node;
Node* parent;
bool  nodeIsLeftChild;

bool move (bool moveLeft);
bool moveRedIfPossible (bool moveLeft);
};
```

The file "tree.hpp"

```
#include "token.hpp"
class Tree {
public:

Node* root;

Node* rotateLeft  (Node* node, Node* parent, bool nodeIsLeftChild);
Node* rotateRight (Node* node, Node* parent, bool nodeIsLeftChild);
```

```
Node* rotate      (bool leftEdge, Node* node, Node* parent, bool nodeIsLeftChild);
Node* rotateToken (bool leftEdge, Token* token);

void  removeLeaf  (Node* parent, bool removeLeftChild);
void  colorFlip   (Node* node);
...
void  deleteLeaf(int key);
}
```

The following code represents the main function of the algorithm. The variable `key` is the key of the leaf to be deleted.

```
void Tree ::  deleteLeaf(int key) {
|  Token* token = new Token(this->root);
|  while(true) {
|  |  // 1st PHASE - find an output node of the actual cluster
|  |  if(token->node->key != key) token->moveRedIfPossible(key < token->node->key);
|  |  // if the target leaf is reached, delete it and exit
|  |  if(token->node->key == key) {this->remove(token->parent,token->nodeIsLeftChild); return;}
|  |  // 2nd PHASE - make the output node a cluster leaf
|  |  bool pathGoesLeft = key < token->node->key;
|  |  if(token->node->hasRedChild(!pathGoesLeft)) {
|  |  |  token->node = this->rotateToken(!pathGoesLeft,token);
|  |  |  token->move(pathGoesLeft);
|  |  }
|  |  // 3rd PHASE
|  |  Node* node = token->node;
|  |  Node* nextRoot = node->child(pathGoesLeft);
|  |  Node* sibling = node->child(!pathGoesLeft);
|  |  if(nextRoot->hasRedChildren()) {
|  |  |  // the cluster rooted in nextRoot is delete-ready, move to its root
|  |  |  token->move(pathGoesLeft);
|  |  }
|  |  else {
|  |  |  // join the nextRoot cluster with the sibling cluster
|  |  |  this->colorFlip(node);
|  |  |  if(sibling->hasRedChildren()) {
|  |  |  |  // remove possible zig-zag
|  |  |  |  if(!sibling->hasRedChild(!pathGoesLeft)) this->rotate(pathGoesLeft,sibling,node,!pathGoesLeft);
|  |  |  |  // make the search path branch of the joined cluster bigger by a rotation
|  |  |  |  token->node = this->rotateToken(!pathGoesLeft,token);
|  |  |  |  // split back the joined cluster
|  |  |  |  this->colorFlip(token->node);
|  |  |  |  // move the token to the root of the search path cluster
|  |  |  |  token->move(pathGoesLeft);
|  |  |  }
|  |  }
|  }
}
```

Taking into account the previous analysis of the problem, there is hopefully no need of further explanation of the code.

9 Experimental results

In this preliminary report, only one simple result is presented. 1,000,000 experiments were executed with a deletion of a random leaf from a random tree with 2047 nodes (the size of the largest tree of the depth 10 - measured by the number of edges of the longest path from the root to a leaf).

We use the following code of an execution of an iteration of the algorithm: the code is composed of two letters

the first letter shows how the first and the second phases are executed:

- **0** both the first and the second phase are void;
- **M** the first phase consists of one token move to a red node, the second phase is void;
- **R** the first phase is void, the second phase consists of one rotation of an edge connecting the token node with its off-path red child;

the second letter says how the third phase is executed:

- **0** one token move to the root of a delete-ready cluster;
- **1** a cluster join, a rotation, a cluster split, and a token move
- **2** a cluster join, a pre-rotation to remove a zig-zag, a rotation, a cluster split, and a token move
- **J** just a cluster join of two one-node clusters with the token node

In the above mention experiment, we have found that

the average tree depth was 14.6
the average number of iterations per one delete operation was 8.0

and percentual number of different types of iterations was

Type	00	01	02	0J	M0	M1	M2	MJ	R0	R1	R2	RJ	D
Percentage	6.1	2.4	0.7	0.3	31.7	11.7	4.4	14.5	9.0	1.2	0.9	4.6	12.5
Rotations	0	1	2	0	0	1	2	0	1	2	3	1	0
Color flips	0	2	2	0	0	2	2	0	0	2	2	0	0

where the column D represents iterations that finished in the first phase by deletion of a leaf.

Combining together, the percentage of different executions of the first two phases was

The type 0 9.4 %
The type M 62.4 %
The type R 15.7 %

The percentage of different executions of the third phase was

The type 0 46.8 %
The type 1 19.0 %
The type 2 6.0 %
The type J 19.4 %

On the average, when applying a `deleteLeaf` to a randomly constructed tree with 2047 nodes (the average of the depths was 14.6),
the average of the number of rotations per one deletion was 3.45, and
the average of the number of color flips per one deletion was 1.93.

References

[1] www.algovision.org/Algovision/index.html

[2] R. Bayer, *Symmetric binary B-trees: data structure and maintenance algorithms*, Acta Informatica I (1972), 290-306.

[3] T. H. Corman, C. E. Leiserson, R. L. Rivest and C. Stein, Introduction to Algorithms, 2nd edition, MIT Press 2001, pp. 273-301.

[4] L. Guibas and R. Sedgewick, *A dichromatic framework for balanced trees*, Proceedings of the 19th Annual Conference on Foundations of Computer Science, Ann Arbor, MI (1978).

[5] R. Sedgewick Left Leaning Red-Black Trees, Princeton University, www.cs.princeton.edu/∼rs/talks/LLRB/LLRB.pdf

10 Appendix

In this appendix an implementation of header files `node.hpp`, `token.hpp` and `tree.hpp` are given to present a complete formal description of the algorithm.

The file `node.cpp`:

```
#include "node.hpp"
using namespace std;

Node :: Node(int key) {
this->leftChild  = NULL;
this->rightChild = NULL;
this->key        = key;
this->isRed      = true;
}

bool Node :: hasRedLeftChild () {
return (this->leftChild  != NULL) && this->leftChild->isRed;
}

bool Node :: hasRedRightChild () {
return (this->rightChild != NULL) && this->rightChild->isRed;
}

bool Node :: hasRedChild (bool lookForLeftChild) {
if(lookForLeftChild) return this->hasRedLeftChild();
else                 return this->hasRedRightChild();
}

bool Node :: hasRedChildren () {
return this->hasRedChild(true) || this->hasRedChild(false);
}

Node* Node :: child(bool leftChildRequested) {
if(leftChildRequested) return this->leftChild;
else                   return this->rightChild;
}

void Node :: setChild(bool asLeftChild, Node* child) {
if(asLeftChild) this-> leftChild  = child;
else            this-> rightChild = child;
}
```

The file `token.cpp`:

```
#include "token.hpp"
using namespace std;

Token :: Token(Node* node) {
this->node = node;
this->parent = NULL;
}
```

```
bool Token :: move (bool moveLeft) {
Node *node = this->node;
if(moveLeft) {
if(node->leftChild != NULL) {
this->parent = node;
this->node = node->leftChild;
this->nodeIsLeftChild = true;
return true;
}
else return false;
} else {
if(node->rightChild != NULL) {
this->parent = node;
this->node = node->rightChild;
this->nodeIsLeftChild = false;
return true;
}
else return false;
}
}

bool Token :: moveRedIfPossible (bool moveLeft) {
if(this->node->hasRedChild(moveLeft)) return move(moveLeft); else return false;
}
```

The file `tree.cpp`:

```
#include "tree.hpp"
using namespace std;

Node* Tree :: rotateLeft (Node* node, Node* parent, bool nodeIsLeftChild) {
Node* child = node->leftChild;
Node* grandChild = child->rightChild;
node->leftChild = grandChild;
child->rightChild = node;
if(parent != NULL) parent->setChild(nodeIsLeftChild,child);
else this->root = child;
child->isRed = node->isRed;
node->isRed = true;
return child;
}

Node* Tree :: rotateRight (Node* node, Node* parent, bool nodeIsLeftChild) {
Node* child = node->rightChild;
Node* grandChild = child->leftChild;
node->rightChild = grandChild;
child->leftChild = node;
if(parent != NULL) parent->setChild(nodeIsLeftChild,child);
else this->root = child;
child->isRed = node->isRed;
node->isRed = true;
return child;
}

Node* Tree :: rotate (bool leftEdge, Node* node, Node* parent, bool nodeIsLeftChild) {
```

```
if(leftEdge) return rotateLeft (node,parent,nodeIsLeftChild);
else         return rotateRight(node,parent,nodeIsLeftChild);
}

Node* Tree :: rotateToken (bool leftEdge, Token* token) {
return rotate(leftEdge,token->node,token->parent,token->nodeIsLeftChild);
}

void Tree :: removeLeaf (Node* parent, bool removeLeftChild) {
if(parent != NULL) parent->setChild(removeLeftChild,NULL);
else this->root = NULL;
}

void Tree :: colorFlip (Node* node) {
if(node != this->root) node->isRed = !node->isRed;
node->leftChild ->isRed = !node->leftChild ->isRed;
node->rightChild->isRed = !node->rightChild->isRed;
}
```