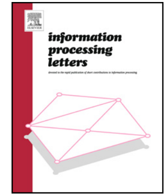
Information Processing Letters 114 (2014) 591–596

Contents lists available at ScienceDirect

Information Processing Letters

www.elsevier.com/locate/ipl



# Red-black trees with relative node keys

Mike Holenderski *, Reinder J. Bril, Johan J. Lukkien

*Eindhoven University of Technology, Den Dolech 2, 5612AZ Eindhoven, The Netherlands*

ARTICLE INFO

*Article history:*
Received 7 February 2014
Received in revised form 6 May 2014
Accepted 9 June 2014
Available online 11 June 2014
Communicated by M. Chrobak

*Keywords:*
Data structures
Algorithms
Search trees
Red-black trees
Relative keys

ABSTRACT

This paper addresses the problem of storing an ordered list using a red-black tree, where node keys can only be expressed relative to each other. The insert and delete operations in a red-black tree are extended to maintain the relative key values. The extensions rely only on relative keys of neighboring nodes, adding constant overhead and thus preserving the logarithmic time complexity of the original operations.

© 2014 Elsevier B.V. All rights reserved.

## 1. Introduction

Red-black trees [1,2] are data structures which provide a logarithmic time bound for retrieving, inserting and deleting a node from an ordered list based on its key. The key of a node is usually stored as an *absolute* value, allowing to directly compare the keys of two arbitrary nodes in the tree. In some cases, however, one may need to store the key values *relative* to other keys in the tree [3]. If storing both relative and absolute keys is not possible (e.g. due to memory limitations), then one must be able to reconstruct the absolute keys from the relative keys. The relative keys must therefore be maintained during any operation which modifies the tree structure. This paper extends the insertion and deletion operations in a red-black tree to maintain the relative node keys, while preserving the logarithmic time complexity of the original operations.

## 2. Preliminaries

Let $n$ be a node in a list and $a(n)$ its *absolute* key value. Let $(n_1, n_2, \dots, n_N)$ be a list of $N$ nodes ordered by their keys, i.e. $\forall i : 1 \le i < N : a(n_i) \le a(n_{i+1})$. We use a red-black tree as an underlying data structure for the ordered list. Given a particular tree we define:

- $parent(n)$ is the parent of node $n$ (if it exists).
- $left(n)$ is the left child of node $n$ (if it exists).
- $right(n)$ is the right child of node $n$ (if it exists).
- $root$ is the root node of the tree.
- $r(n)$ is the *relative* key value of node $n$, defined as follows:

$$a\big(left(n)\big) = a(n) - r\big(left(n)\big) \tag{1}$$

$$a\big(right(n)\big) = a(n) + r\big(right(n)\big) \tag{2}$$

Note that the *root* is neither a left nor a right child, so its relative key is not defined. The relative key representation is illustrated in Fig. 1.

* Corresponding author.
*E-mail addresses:* m.holenderski@tue.nl (M. Holenderski), r.j.bril@tue.nl (R.J. Bril), j.j.lukkien@tue.nl (J.J. Lukkien).

http://dx.doi.org/10.1016/j.ipl.2014.06.004
0020-0190/© 2014 Elsevier B.V. All rights reserved.

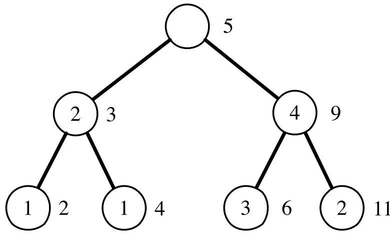



**Fig. 1.** Example of storing an ordered list $(2, 3, 4, 5, 6, 9, 11)$ using a red-black tree with relative keys. The numbers inside and outside the nodes represent $r(n)$ and $a(n)$ values, respectively.

We will use the shorthand notation:

$$isLeft(n) \equiv n = left(parent(n))$$

$$isRight(n) \equiv n = right(parent(n))$$

$$isRoot(n) \equiv n = root$$

## 3. Design

For each node $n$ (with exception of the root) we only store its relative key $r(n)$. Let *n's root path* be the shortest path between the root and node $n$. Assuming we know the absolute key of the root node, we can compute the absolute key $a(n)$ for any node $n$ by starting at the root and accumulating the relative keys along its root path according to (1) and (2). We store and maintain the absolute key of the root node in the variable *aRoot*.

When inserting or deleting nodes in our red-black tree we need to maintain the relative keys stored in the nodes in such a way that after performing the operation the root paths for all nodes will have the same value as they had before the operation. Let $p(n, t)$ be the value of the root path of node $n$ in tree $t$, computed recursively according to (1) and (2). Let $T$ and $T'$ denote the tree before and after performing an insert or delete operation, respectively. For both operations we will need to prove that the following invariant is maintained:

$$\forall n \in T \cap T': \quad p(n, T') = p(n, T) = a(n). \tag{3}$$

Note that (3) must hold only for nodes which are in the tree before and after the operation (i.e. we can exclude the inserted or deleted node). For the inserted node $n$ we must have $p(n, T') = a(n)$. For nodes which are not in the tree we assume that any predicate on them holds, i.e.

$$\forall x, T, P: \quad x \notin T \Rightarrow P(x, T) \equiv true \tag{4}$$

where $x$ is a node, $T$ is a tree, and $P$ is a predicate.

## 4. Insert

Inserting a value into a red-black tree involves the following steps [1,2]:

1. find a leaf position for inserting the value,
2. add a new leaf node,
3. balance the tree.

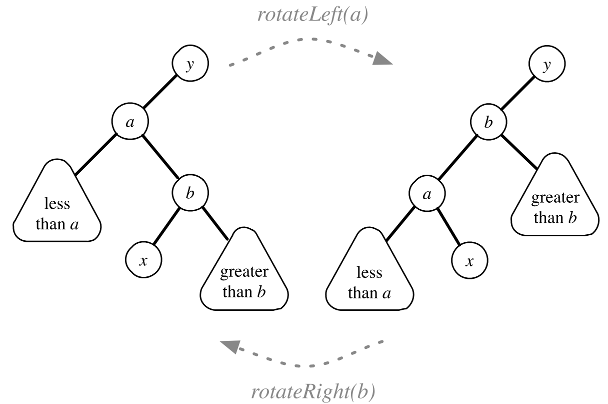



**Fig. 2.** Instances of rotation operations when $isLeft(a)$ holds. The "less than $a$" subtree contains nodes with smaller absolute keys than $a(a)$. Node $x$ may or may not exist. Notice that $a$, $b$, $x$ and $y$ are labels, rather than relative keys in Fig. 1.

Step 1 traverses the tree looking for the right position to insert a new node for the inserted value. While traversing the tree the values of the root paths can be computed by accumulating the relative keys stored in the nodes according to (1) and (2). Adding a leaf node in step 2 does not impact the relative keys of any other nodes in the tree. Therefore, relative keys do not need to be adapted in steps 1 and 2.

Step 3 relies on two rotation operations, *rotateLeft*() and *rotateRight*(), to rebalance the tree. During these operations, the relative position of nodes in the tree is changed, and consequently the relative keys of the affected nodes need to be adapted. In this section we show how to extend these two operations to maintain invariant (3).

### 4.1. Rotate left

In our red-black tree we store only the relative keys in the nodes, which satisfy (1) and (2). Therefore, to maintain invariant (3), after any operation which modifies the tree structure by adding, removing or rotating nodes, the relative keys must be updated. We distinguish three cases, depending on whether (i) $a$ is a left child, (ii) $a$ is a right child, or (iii) $a$ is the root.

#### 4.1.1. Case: a is the left child of its parent

The $rotateLeft(a)$ operation when $isLeft(a)$ is illustrated in Fig. 2. The original $rotateLeft(a)$ operation can be defined in terms of the following pre and post condition, using notation in [4]:

$$\{P\}$$
$$rotateLeft(a);$$
$$\{Q\} \tag{5}$$

where

$$P: a = left(y) \wedge b = right(a) \wedge x = left(b)$$

$$Q: a = left(b) \wedge b = left(y) \wedge x = right(a)$$

Notice that in $P$ and $Q$ we only list the conditions which are affected by the *rotateLeft*() operation and for brevity we omit the unaffected ones.

To maintain invariant (3), we need to modify the relative keys stored in the nodes. We propose the following parallel assignment:

$$r(a), r(b), r(x) := r(b), r(a) - r(b), r(b) - r(x) \tag{6}$$

To prove that (3) is indeed maintained, we need to show that the values of the root paths of all the nodes in the tree are the same as before the *rotateLeft*() operation. In particular, we need to show that we can derive the same root path value for all nodes which have acquired a new parent. In Fig. 2 these are nodes $a$, $b$, and $x$.

**Lemma 1.** *When isLeft*$(a)$ *holds, extending operation rotateLeft*$(a)$ *with assignment* (6) *maintains invariant* (3).

**Proof.** From (1), (2) and (5) we can derive the following pre and post conditions for (6) relating the absolute and relative keys before and after *rotateLeft*$(a)$ when *isLeft*$(a)$:

$$\{R\}$$
$$r(a), r(b), r(x) := r(b), r(a) - r(b), r(b) - r(x);$$
$$\{S\}$$

where

$$R : a(a) = a(y) - r(a) \wedge a(b) = a(a) + r(b) \wedge a(x) = a(b) - r(x)$$
$$S : a(a) = a(b) - r(a) \wedge a(b) = a(y) - r(b) \wedge a(x) = a(a) + r(x)$$

We need to show that $R \Rightarrow S(r(a), r(b), r(x) := r(b), r(a) - r(b), r(b) - r(x))$.

$$\begin{aligned}
&S\big(r(a), r(b), r(x) := r(b), r(a) - r(b), r(b) - r(x)\big) \\
&\quad \equiv \{\text{definition of } S\} \\
&a(a) = a(b) - r(b) \wedge a(b) = a(y) - r(a) + r(b) \\
&\qquad \wedge a(x) = a(a) + r(b) - r(x) \\
&\quad \equiv \{\text{arithmetic}\} \\
&a(a) = a(y) - r(a) \wedge a(b) = a(a) + r(b) \wedge a(x) \\
&\quad = a(b) - r(x) \\
&\quad \equiv \{\text{definition of } R\} \\
&R
\end{aligned}$$

□

*4.1.2. Case: a is the right child of its parent*

The *rotateLeft*$(a)$ operation when *isRight*$(a)$ can be specified as follows:

$$\{P\}$$
$$rotateLeft(a);$$
$$\{Q\} \tag{7}$$

**Algorithm 1** Extended *rotateLeft*().

```
procedure ROTATELEFT'(a)
    b := right(a);
    x := left(b);
    if isLeft(a) then
        r(a), r(b), r(x) := r(b), r(a) − r(b), r(b) − r(x);
    else if isRight(a) then
        r(a), r(b), r(x) := r(b), r(a) + r(b), r(b) − r(x);
    else
        aRoot, r(a), r(x) := aRoot + r(b), r(b), r(b) − r(x);
    end if
    rotateLeft(a);
end procedure
```

where

$$P : a = right(y) \wedge b = right(a) \wedge x = left(b)$$
$$Q : a = left(b) \wedge b = right(y) \wedge x = right(a)$$

We can show in a similar way to the case when *isLeft*$(a)$, that invariant (3) will be maintained for *isRight*$(a)$ if we extend the *rotateLeft*$(a)$ operation with assignment

$$r(a), r(b), r(x) := r(b), r(a) + r(b), r(b) - r(x) \tag{8}$$

Notice the subtle difference in the assignment to $r(b)$.

*4.1.3. Case: a is the root*

The *rotateLeft*$(a)$ operation when *isRoot*$(a)$ can be specified as follows:

$$\{a = root \wedge b = right(a) \wedge x = left(b)\}$$
$$rotateLeft(a);$$
$$\{a = left(b) \wedge b = root \wedge x = right(a)\}$$

Since in this case $a$ is the root of the tree, it does not have a parent. As mentioned in Section 3, we must therefore maintain the absolute key of the root *aRoot* directly. We can show in a similar way to the case when *isLeft*$(a)$, that invariant (3) will be maintained when *isRoot*$(a)$ holds if we extend the *rotateLeft*$(a)$ operation with assignment

$$aRoot, r(a), r(x) := aRoot + r(b), r(b), r(b) - r(x)$$

Notice that after the rotation node $b$ becomes the root. Since $b$ does not have a parent and $a(b) = aRoot$, its relative key $r(b)$ can be arbitrary (in our case it simply remains the same).

The extended *rotateLeft*() operation is summarized in Algorithm 1.

*4.2. Rotate right*

The extended *rotateRight*() operation is summarized in Algorithm 2. It can be shown in a way similar to Section 4.1 that the proposed extension maintains invariant (3).

## 5. Deleting nodes

Deleting a node $n$ from a red-black tree involves the following steps [1,2]:

**Algorithm 2** Extended *rotateRight*().

```
procedure ROTATERIGHT'(b)
  a := left(b);
  x := right(a);
  if isLeft(b) then
    r(a), r(b), r(x) := r(b) + r(a), r(a), r(a) − r(x);
  else if isRight(b) then
    r(a), r(b), r(x) := r(b) − r(a), r(a), r(a) − r(x);
  else
    aRoot, r(b), r(x) := aRoot − r(a), r(a), r(a) − r(x);
  end if
  rotateRight(b);
end procedure
```

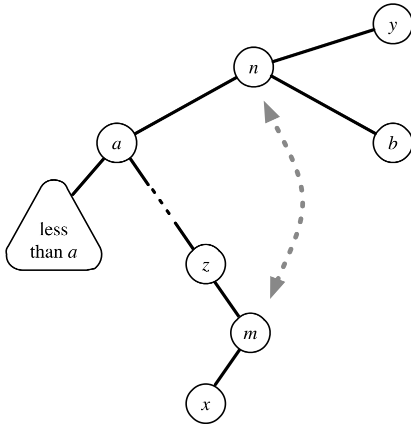



**Fig. 3.** The $swap(n, m)$ operation in a red-black tree, when $isLeft(n)$ and $a \neq m$ hold. Node $x$ may or may not exist and it is possible that $a = z$.

1. find node $m$ which is $n$'s predecessor in the ordered list,[1]
2. swap $n$ with $m$,
3. rebalance the tree,
4. remove $n$.

Steps 2, 4 and 3 modify the tree structure. Step 3 relies on rotation operations to rebalance the tree, which we have already addressed in Section 4. In this section we show how to extend the *swap*() and *remove*() operations in steps 2 and 4 to maintain invariant (3).

### 5.1. Swap

Fig. 3 illustrates the swap operation. A swap is performed only if $n$ has two children, therefore, we can assume that nodes $a$ and $b$ in Fig. 3 exist. Node $x$ represents the root of the subtree with nodes smaller than $m$. This subtree, and thus also $x$, may or may not exist. Since $m$ is the predecessor of $n$ in the ordered list (i.e. the largest node in the subtree rooted at $a$), $m$ cannot have a right child.

We first consider the case when $a \neq m$, illustrated in Fig. 3. Notice that $z$ represents the parent node of $m$ and it is very well possible that $a = z$. We have introduced it here only to be able to express the absolute key of $m$ relative to its parent when showing that (3) holds. Similar to Section 4 we distinguish three cases, depending on whether (i) $a$ is a left child, (ii) $a$ is a right child, or (iii) $a$ is the root.

[1] Some red-black tree implementations search for $n$'s successor node [5,6], resulting in slightly different extension of the *swap*() operation. However, the correctness of the extension can be shown similar to the argument presented in this section.

#### 5.1.1. Case: a is the left child of its parent

The original $swap(a)$ operation can be defined in terms of the following pre and post condition:

$$\{P\}$$
$$swap(n, m);$$
$$\{Q\} \tag{9}$$

where

$$P : n = left(y) \wedge m = right(z) \wedge x = left(m) \wedge a = left(n) \wedge b = right(n)$$
$$Q : m = left(y) \wedge n = right(z) \wedge x = left(n) \wedge a = left(m) \wedge b = right(m)$$

Adjusting the relative keys when swapping nodes $n$ and $m$ requires the knowledge of $d(n, m)$, which represents the difference between $a(n)$ and $a(m)$, i.e

$$d(n, m) = a(n) - a(m) \tag{10}$$

Let us introduce the following shorthand notation:

$$R_1 : a(n) = a(y) - r(n) \qquad S_1 : a(m) = a(y) - r(m)$$
$$R_2 : a(m) = a(z) + r(m) \qquad S_2 : a(n) = a(z) + r(n)$$
$$R_3 : a(x) = a(m) - r(x) \qquad S_3 : a(x) = a(n) - r(x)$$
$$R_4 : a(a) = a(n) - r(a) \qquad S_4 : a(a) = a(m) - r(a)$$
$$R_5 : a(b) = a(n) + r(b) \qquad S_5 : a(b) = a(m) + r(b)$$

**Lemma 2.**

$$\{P : R_1 \wedge R_2 \wedge R_3 \wedge R_4 \wedge R_5\}$$
$$r(n), r(m) := r(m) + d(n, m), r(n) + d(n, m);$$
$$\{Q : S_1 \wedge S_2 \wedge R_3 \wedge R_4 \wedge R_5\}$$

**Proof.** We need to show that

$$P \Rightarrow Q\big(r(n), r(m) := r(m) + d(n, m), r(n) + d(n, m)\big).$$

$$Q\big(r(n), r(m) := r(m) + d(n, m), r(n) + d(n, m)\big)$$
$$\equiv \{\text{definition of } S_1, S_2, R_3, R_4, R_5\}$$
$$a(m) = a(y) - r(n) - d(n, m) \wedge a(n) = a(z) + r(m) + d(n, m) \wedge R_3 \wedge R_4 \wedge R_5$$
$$\equiv \{\text{arithmetic, (10)}\}$$
$$a(n) = a(y) - r(n) \wedge a(m) = a(z) + r(m) \wedge R_3 \wedge R_4 \wedge R_5$$
$$\equiv \{\text{definition of } R_1, R_2\}$$
$$P \qquad \square$$

**Lemma 3.**

$\{P : S_1 \wedge S_2 \wedge R_3 \wedge R_4 \wedge R_5\}$

**if** $\exists x : x = \mathit{left}(m)$ **then**

$r(x) := r(x) + d(n,m);$

**end if**

$\{Q : S_1 \wedge S_2 \wedge S_3 \wedge R_4 \wedge R_5\}$

**Proof.** We need to show that the following two conditions hold:

$$\big(P \wedge \exists x : x = \mathit{left}(m)\big) \Rightarrow Q\,\big(r(x) := r(x) + d(n,m)\big) \quad (11)$$

$$\big(P \wedge \neg\exists x : x = \mathit{left}(m)\big) \Rightarrow Q \quad (12)$$

Eq. (12) follows from (4). Now we show that (11) holds:

$$\begin{aligned}
& Q\,\big(r(x) := r(x) + d(n,m)\big) \\
& \quad \equiv \{\text{definition of } S_1, S_2, S_3, R_4, R_5\} \\
& S_1 \wedge S_2 \wedge a(x) = a(n) - r(x) - d(n,m) \wedge R_4 \wedge R_5 \\
& \quad \equiv \{\text{arithmetic, (10)}\} \\
& S_1 \wedge S_2 \wedge a(x) = a(m) - r(x) \wedge R_4 \wedge R_5 \\
& \quad \equiv \{\text{definition of } R_3\} \\
& P
\end{aligned}$$

□

**Lemma 4.**

$\{P : S_1 \wedge S_2 \wedge S_3 \wedge R_4 \wedge R_5\}$

$r(a) := r(a) - d(n,m);$

$\{Q : S_1 \wedge S_2 \wedge S_3 \wedge S_4 \wedge R_5\}$

**Lemma 5.**

$\{P : S_1 \wedge S_2 \wedge S_3 \wedge S_4 \wedge R_5\}$

$r(b) := r(b) + d(n,m);$

$\{Q : S_1 \wedge S_2 \wedge S_3 \wedge S_4 \wedge S_5\}$

**Lemma 6.** *When isLeft(n) and $a \neq m$ hold, extending operation swap(n, m) with program E given by*

$r(n), r(m) := r(m) + d(n,m), r(n) + d(n,m);$

**if** $\exists x : x = \mathit{left}(m)$ **then**

$r(x) := r(x) + d(n,m);$

**end if**

$r(a) := r(a) - d(n,m);$

$r(b) := r(b) + d(n,m);$

*where $d(n,m) = a(n) - a(m)$, maintains invariant (3).*

**Algorithm 3** Extended *swap*().

```
procedure SWAP'(n, m, d)
    a := left(n);
    b := right(n);
    if a ≠ m then
        if isLeft(n) then
            r(n), r(m) := r(m) + d, r(n) + d;
        else if isRight(n) then
            r(n), r(m) := r(m) + d, r(n) − d;
        else
            r(n), aRoot := r(m) + d, aRoot − d;
        end if
        r(a) := r(a) − d;
    else
        if isLeft(n) then
            r(n), r(m) := −r(m), r(n) + d;
        else if isRight(n) then
            r(n), r(m) := −r(m), r(n) − d;
        else
            r(n), aRoot := −r(m), aRoot − d;
        end if
    end if
    if ∃x : x = left(m) then
        r(x) := r(x) + d;
    end if
    r(b) := r(b) + d;
    swap(n, m);
end procedure
```

**Proof.** From (1), (2) and (9) we can derive the following pre and post conditions for the proposed extension $E$:

$\{R_1 \wedge R_2 \wedge R_3 \wedge R_4 \wedge R_5\}$

$E$

$\{S_1 \wedge S_2 \wedge S_3 \wedge S_4 \wedge S_5\}$

From Lemmas 2, 3, 4 and 5, and their sequential composition it follows that $E$ indeed satisfies the above pre and post conditions. □

#### 5.1.2. The other cases

For the other cases when $a \neq m$ (i.e. when $n$ is the right child of its parent or when $n$ is the root) and for the case when $a = m$, we can follow a similar argument to prove that (3) is maintained by the *swap*() extension summarized in Algorithm 3. The $d$ parameter represents $d(n,m)$ which can be accumulated while searching for $m$ in step 1 of the delete operation. Searching for $m$ is limited to traversing the nodes on $m$'s root path and therefore has logarithmic time complexity. Accumulating $d$ adds a constant overhead.[2]

### 5.2. Remove

The delete operation in a red-black tree guarantees that when *remove*($n$) is called node $n$ either has no children, or only one child, indicated by the $x$ node in Fig. 4. The *remove*($n$) operation removes node $n$ and substitutes it by its child $x$, if any. It can be shown similar to Sections 4.1 and 5.1 that the extension of *remove*($n$) shown in Algorithm 4 maintains invariant (3).

[2] During the *swap*($n, m$) operation, $r(n)$ can become negative. Node $n$, however, is removed later during the *remove*($n$) operation. It is the only place and time when a node can have a negative relative key value.

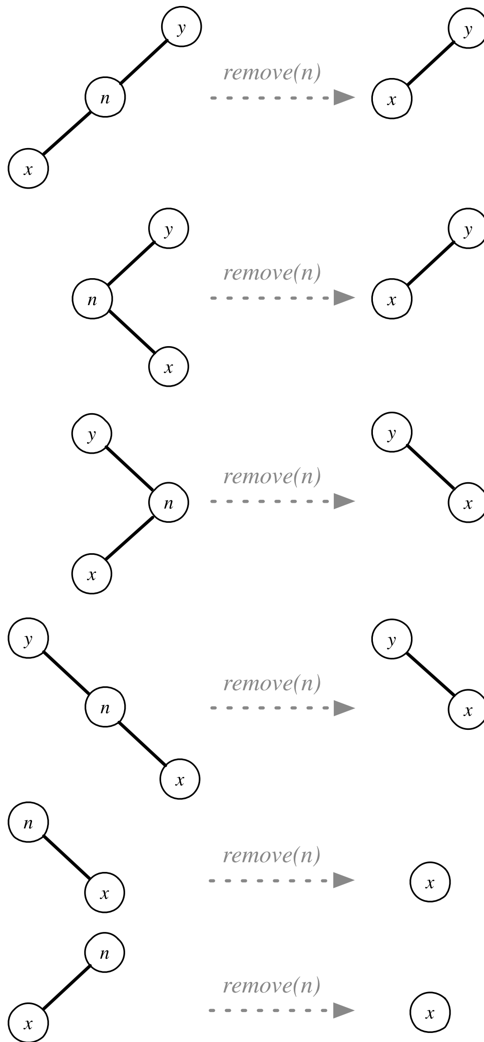



**Fig. 4.** Six cases distinguishing the relative positions of $n$ and its child $x$ with respect to their parents when removing $n$ during a delete operation in a red-black tree.

**Algorithm 4** Extended *remove*().

```
procedure REMOVE'(n)
    if ∃x : x = left(n) then
        if (isLeft(n) ∧ isLeft(x)) ∨ (isRight(n) ∧ isRight(x)) then
            r(x) := r(n) + r(x);
        else if (isLeft(n) ∧ isRight(x)) ∨ (isRight(n) ∧ isLeft(x)) then
            r(x) := r(n) − r(x);
        else
            if isLeft(x) then
                aRoot := aRoot − r(x);
            else if isRight(x) then
                aRoot := aRoot + r(x);
            end if
        end if
    end if
    remove(n);
end procedure
```

## 6. Discussion

There are several well known data structures exhibiting logarithmic insertion and deletion time complexity, such as red-black trees, AVL trees, and certain implementations of the heap [5]. In this paper we focused on extending the red-black tree with relative keys. The presented approach can be easily applied to balanced trees which rely on rotations for balancing, such as AVL trees. Balanced tree implementations that rely on other operations for balancing, such as the array implementation of a heap, can be extended in a similar way, relying on and updating only the relative keys residing locally in the immediate relatives (parent, siblings and children).

In a memory constrained environment we like to limit the representation of the keys (e.g. limit to 16 bits on embedded platforms). Inserting a node with a key much larger than the largest representable key value is made possible by the relative key representation. However, it will require inserting intermediate nodes [3]. In the standard array-based heap implementation, adding such 'dummy' keys requires filling an entire layer in the tree while this impact is much better amortized for rotation-based trees. Some further study is required to highlight the detailed differences.

## 7. Conclusion

We have shown how to support relative keys in a red-black tree by extending the methods which modify the tree structure when inserting or deleting nodes. The extensions are simple assignments adapting the relative keys stored in the nodes. The assignments require only information which is available locally, i.e. the relative keys of the neighboring nodes in the tree. Therefore, the logarithmic time complexity of the insert and delete operations in a red-black tree is maintained.

## Acknowledgements

We would like to thank Martijn van den Heuvel for motivation and insightful discussions.

## References

[1] R. Bayer, Symmetric binary b-trees: data structure and maintenance algorithms, Acta Inform. 1 (4) (1972) 290–306.
[2] L.J. Guibas, R. Sedgewick, A dichromatic framework for balanced trees, in: 19th Annual Symposium on Foundations of Computer Science, 1978, pp. 8–21.
[3] M. Holenderski, R.J. Bril, J.J. Lukkien, An efficient hierarchical scheduling framework for the automotive domain, in: S.M. Babamir (Ed.), Real-Time Systems, Architecture, Scheduling, and Application, InTech, 2012, pp. 67–94.
[4] C.A.R. Hoare, An axiomatic basis for computer programming, Commun. ACM 12 (10) (1969) 576–580.
[5] T.H. Cormen, C. Stein, R.L. Rivest, C.E. Leiserson, Introduction to Algorithms, 2nd edition, McGraw-Hill Higher Education, 2001.
[6] R. Sedgewick, Left-leaning red-black trees, Tech. rep. Princeton University, 2008, https://www.cs.princeton.edu/~rs/talks/LLRB/LLRB.pdf.