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.

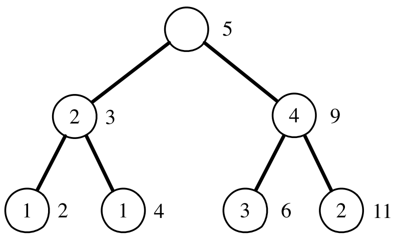

Fig. 1. Example of storing an ordered list $(2, 3, 4, 5, 6, 9, 11)$ using a red-black tree with relative keys. The numbers inside and outside the nodes represent $r(n)$ and $a(n)$ values, respectively.

We will use the shorthand notation:

$$isLeft(n) \equiv n = left\big(parent(n)\big)$$
$$isRight(n) \equiv n = right\big(parent(n)\big)$$
$$isRoot(n) \equiv n = root$$

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:

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

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.

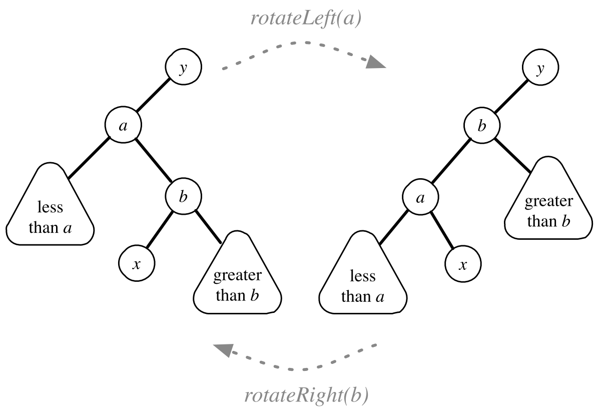

Fig. 2. Instances of rotation operations when $isLeft(a)$ holds. The "less than a" subtree contains nodes with smaller absolute keys than $a(a)$. Node x may or may not exist. Notice that a, b, x and y are labels, rather than relative keys in Fig. 1.

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

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