

**Dati e Algoritmi I (Pietracaprina)**

**Removal in Binary Search Trees  
and (2,4)-Trees**

## 1 Introduction

The purpose of this note is to describe the implementation of method `remove` of the Dictionary ADT ([GT06, Sec. 9.3]) on Binary Search Trees and (2,4)-Trees. The implementations presented in [GT06, Ch. 10] refer to a different specification of the method, which was employed in previous editions of the book, hence they are not consistent with the current specification.

Recall that the invocation `remove( $e$ )` for a dictionary  $D$  removes and returns  $e$ , if  $e$  is in  $D$ , otherwise it returns `null` and leaves  $D$  unchanged.

## 2 Binary Search Tree implementation

Let  $T$  be a Binary Search Tree implementing a dictionary with  $n$  entries. Let also `EntrySearch( $e, v$ )` be an algorithm that returns a node  $w \in T_v$ , where  $w$  is internal and contains entry  $e$ , if  $e$  is stored in  $T_v$ , otherwise  $w$  is a leaf<sup>1</sup>. Such an algorithm will be described later. The pseudocode of method `remove` is reported in Figure 1.

---

Figure 1: Implementation of `remove( $e$ )` for Binary Search Trees

---

```

 $w \leftarrow$  EntrySearch( $e, T.root()$ );
if ( $T.isExternal(w)$ ) then return null;
if ( $T.isExternal(T.left(w))$ ) then
    replace  $w$  with its right child;
    decrease the size of  $T$  by 1;
    return  $e$ ;
if ( $T.isExternal(T.right(w))$ ) then
    replace  $w$  with its left child;
    decrease the size of  $T$  by 1;
    return  $e$ ;
 $u \leftarrow$  rightmost internal node in  $w$ 's left subtree;
put  $u.element()$  into  $w$ ;
replace  $u$  with its left child;
decrease the size of  $T$  by 1;
return  $e$ ;

```

---

If both children of the node  $w$  returned by `EntrySearch` are internal,  $e$  cannot be simply removed from  $w$ , but it is substituted in  $w$  with its predecessor  $e'$  in the key-based ordering, and  $e'$  is removed from node  $u$  where it resides, which is the internal predecessor of  $w$  in

---

<sup>1</sup> $T_v$  denotes the subtree rooted at  $v$ .

the inorder visit of the tree, that is, the rightmost internal node in  $w$ 's left subtree. The complexity of the method, ignoring for now the complexity of `EntrySearch`, is proportional to the height of the tree and is dominated by the search of  $u$  which requires moving from  $w$  down to the bottom of the tree.

Algorithm `EntrySearch` is recursive and is shown in Figure 2. It invokes `TreeSearch(e.getKey(),v)` which returns the first node  $w$  holding an entry with key  $e.getKey()$  encountered in  $T_v$  starting from  $v$ , if such a node exists, or a leaf otherwise. `TreeSearch` is described in [GT06, Sec. 10.1.1].

---

Figure 2: Algorithm `EntrySearch`

---

```

Algorithm EntrySearch( $e, v$ )
input:  entry  $e$ , node  $v \in T$ 
output: node  $w \in T_v$  containing  $e$ , if it exists, or leaf otherwise

 $w \leftarrow$  TreeSearch( $e.getKey(), v$ );
if ( $w.element() = e$ ) OR ( $T.isExternal(w)$ ) then return  $w$ ;
 $u \leftarrow$  EntrySearch( $e, T.left(w)$ );
if ( $T.isInternal(u)$ ) then return  $u$ 
else return EntrySearch( $e, T.right(w)$ );

```

---

The worst-case complexity of `EntrySearch( $e, v$ )` is bounded from above by the height of the subtree  $T_v$  and by the number of entries with key equal to  $e.getKey()$  stored in that subtree. (A rigorous proof of this fact will be obtained as a special case of the analysis of the analogous algorithm developed in the next section for (2,4)-Trees.) Thus, we conclude that the worst-case complexity of method `remove( $e$ )` is

$$\Theta(h + s),$$

where  $h$  is the height of  $T$  and  $s$  the number of entries with key equal to  $e.getKey()$  in  $T$ . Note that this complexity cannot be improved since, if  $e$  is not in  $T$ , time  $\Omega(s)$  is needed to check *all* entries with key equal to  $e.getKey()$  in  $T$  before concluding that  $e$  is not in  $T$ , and, also, time  $\Omega(h)$  is needed since one of these entries could well be at depth  $h$ .

### 3 (2,4)-Tree implementation

The implementation of `remove` on a (2,4)-Tree proceeds in a similar fashion as for the Binary Search Tree, except that extra effort must be devoted to enforce, after the removal of the

entry, the (2,4)-Tree properties. Let  $T$  be a (2,4)-Tree implementing a dictionary with  $n$  entries. Let also  $\text{24EntrySearch}(e, v)$  be an algorithm that returns a node  $w \in T_v$ , where  $w$  is internal and contains entry  $e$ , if  $e$  is stored in  $T_v$ , otherwise  $w$  is a leaf. Such an algorithm will be described later. The pseudocode for method `remove` is reported in Figure 3.

---

Figure 3: Implementation of `remove(e)` for (2,4)-Trees

---

```

w ← 24EntrySearch(e, T.root())
if (T.isExternal(w)) then return null
if (height(w) > 1) then
    let (k1, x1), ..., (kd-1, xd-1) be the entries in w;
    let w1, ..., wd be the children of w;
    let e = (ki, xi) for some 1 ≤ i < d;
    u ← rightmost internal node in subtree Twi;
    let e' be the entry with largest key in u;
    substitute entry e in w with e';
    Delete(e', u);
else Delete(e, w);
decrease the size of T by 1;
return e

```

---

Analogously to the Binary Search Tree case, if the entry  $e$  is held at a node  $w$  whose children are internal (i.e.,  $w$  is at height greater than 1), entry  $e$  cannot be simply removed from  $w$ , but it must be substituted in  $w$  with its predecessor  $e'$  in the key-based ordering, and  $e'$  is removed from node  $u$  where it resides. The actual removal of an entry is performed by algorithm `Delete` which also restores, if needed, the (2,4)-Tree properties. Specifically, the invocation `Delete(e, v)` deletes from  $v$  the entry  $e$  and the child to the right of  $e$  (assuming that this child can be safely deleted), and restores the (2,4)-Tree properties. The way `Delete` can be implemented is described in [GT06, Sec. 10.4.2] and is omitted here for brevity.

The complexity of the method `remove`, ignoring for now the complexity of `24EntrySearch`, is proportional to the height of the tree, which is  $\Theta(\log n)$ . Indeed, both algorithm `Delete` and the search for  $u$  (when  $\text{height}(w) > 1$ ) involve at most a traversal of the tree each, respectively downwards and upwards, performing a constant number of operations at each level.

Let us now discuss algorithm `24EntrySearch`. The algorithm, given in Figure 4, is recursive and calls algorithm `MWTreeSearch(e.getKey(), v)` which works for general multi-way search trees (recall that a (2,4)-Tree is a Multi-Way Search Tree) and returns the first node  $w$  holding an entry with key equal to  $e.getKey()$  encountered in  $T_v$  starting from  $v$ , if such a node exists, or a leaf otherwise. In [GT06, Sec. 10.4.1] the implementation for `MWTreeSearch`

is described, which takes time proportional to the difference  $(\text{depth}(w) - \text{depth}(v))$ , assuming that each node stores  $O(1)$  entries, as is the case for  $(2,4)$ -Trees.

---

Figure 4: Algorithm 24EntrySearch

---

**Algorithm** 24EntrySearch( $e, v$ )

**input:** entry  $e$ , node  $v \in T$

**output:** node  $w \in T_v$  containing  $e$ , if it exists, or leaf otherwise

$w \leftarrow \text{MWTreeSearch}(e.\text{getKey}(), v)$ ;

**if** ( $w$  stores  $e$ ) OR ( $T.\text{isExternal}(w)$ ) **then return**  $w$ ;

let  $(k_1, x_1), \dots, (k_{d-1}, x_{d-1})$  be the entries in  $w$ , and let  $k_0 = -\infty$  and  $k_d = +\infty$ ;

let  $w_1, \dots, w_d$  be the children of  $w$ ;

**for each**  $i$  **such that**  $k_{i-1} \leq e.\text{getKey}() \leq k_i$  **do**

$u \leftarrow \text{24EntrySearch}(e, w_i)$ ;

**if** ( $T.\text{isInternal}(u)$ ) **then break**;

**return**  $u$ ;

---

The time complexity of 24EntrySearch requires a careful analysis. Suppose for simplicity that the algorithm is invoked with  $v = T.\text{root}()$  and let  $k = e.\text{getKey}()$ . In order to get a worst case, we assume that  $e$  is not in  $T$ . The algorithm begins by identifying the first node  $w$  which stores an entry with key equal to  $k$ . This takes  $\Theta(\log n)$  time by invoking MWTreeSearch. At this point, the search for  $e$  continues in a region of  $T_w$  delimited by two paths. One path, say  $L$ , is in a subtree of  $T_w$  containing entries with key  $\leq k$ , the other, say  $R$ , is in a subtree of  $T_w$  containing entries with key  $\geq k$ . All entries found “to the right of path  $L$ ” and all entries found “to the left of path  $R$ ” have key equal to  $k$  and must be checked. An example is shown in Figure 5. This intuition leads us to conclude that the overall running time is obtained by adding up the time to traverse the two paths  $L$  and  $R$ , which is  $\Theta(\log n)$ , and the time to check all entries found to the right of  $L$  and to the left of  $R$ , which is  $\Theta(s)$ , with  $s$  being the number of entries with key  $k$  in the tree.

The above argument can be expressed more rigorously through the following lemmas and the final theorem.

**Lemma 1** *If all entries in  $T_v$  have key equal to  $e.\text{getKey}()$ , then 24EntrySearch( $e, v$ ) takes time proportional to the number of entries in  $T_v$ .*

*Proof.* The algorithm essentially performs a complete visit of  $T_v$  spending at each node time proportional to the entries it contains. □

**Lemma 2** *If all entries in  $T_v$  have keys  $\leq e.\text{getKey}()$ , then  $\text{24EntrySearch}(e, v)$  takes time  $\Theta(h + s)$ , where  $h$  is the height of  $T_v$  and  $s$  is the number of entries with key equal to  $e.\text{getKey}()$  in  $T_v$ .*

*Proof.* We first prove, by induction on  $h$ , that  $\text{24EntrySearch}(e, v)$  takes time at most  $c \cdot (h + s)$ , for a suitably large positive constant  $c$ . The lemma is trivial for  $h = 0, 1$ . Fix  $h \geq 1$  arbitrarily and assume that the lemma holds for trees up to height  $h$ . Consider the invocation of  $\text{24EntrySearch}(e, v)$  with  $v$  at height  $h + 1$ . Let  $k = e.\text{getKey}()$ . First a node  $w$  is found which contains the first occurrence of  $k$  in  $T_v$ . Let  $t \geq 0$  be the distance between  $v$  and  $w$  in  $T_v$ . The search of  $w$  through  $\text{MWTreesearch}$  takes time  $\leq c \cdot (t + 1)$ . Now, all iterations of the **for each** loop, except one, invoke  $\text{24EntrySearch}$  on subtrees containing only entries with key equal to  $k$ . Suppose that there are  $s' \leq s$  entries in these subtrees. By Lemma 1, the time taken by these iterations is  $\leq c \cdot s'$ . Besides these iterations, there is one iteration which invokes  $\text{24EntrySearch}$  on a subtree whose entries have key  $\leq k$ . Such a tree has height at most  $h - t \leq h$  and contains at most  $s - s'$  entries with key equal to  $k$ . By the inductive hypothesis, this iteration takes time  $c \cdot (h - t + s - s')$ . Therefore, the complexity of  $\text{24EntrySearch}(e, v)$  is at most  $c \cdot (h + s) \in O(h + s)$ . To finish the proof we need to show that the complexity of  $\text{24EntrySearch}(e, v)$  is  $\Omega(h + s)$ . To this purpose it is sufficient to observe that if  $e$  does not belong to  $T_v$ , in order to return the correct output  $\text{24EntrySearch}(e, v)$  must reach a leaf of  $T_v$  and must check all entries with the same key as  $e$ .  $\square$

**Lemma 3** *If all entries in  $T_v$  have keys  $\geq e.\text{getKey}()$ , then  $\text{24EntrySearch}(e, v)$  takes time  $\Theta(h + s)$ , where  $h$  is the height of  $T_v$  and  $s$  is the number of entries with key equal to  $e.\text{getKey}()$  in  $T_v$ .*

*Proof.* Similar to the proof of Lemma 2.  $\square$

**Theorem 1**  *$\text{24EntrySearch}(e, v)$  takes time  $\Theta(h + s)$ , where  $h$  is the height of  $T_v$  and  $s$  is the number of entries with key equal to  $e.\text{getKey}()$  in  $T_v$ .*

*Proof.* First the algorithm identifies a node  $w$  containing the first occurrence of  $k$  in  $T_v$ , in time  $\Theta(h)$ . Once  $w$  is found, one iteration of the **for each** loop will invoke  $\text{24EntrySearch}$  on a subtree whose entries have keys  $\leq k$ ; one iteration will invoke  $\text{24EntrySearch}$  on a subtree whose entries have keys  $\geq k$ ; and all other iterations will invoke  $\text{24EntrySearch}$  on subtrees whose entries have key equal to  $k$ . The theorem follows by applying Lemmas 1, 2 and 3.  $\square$

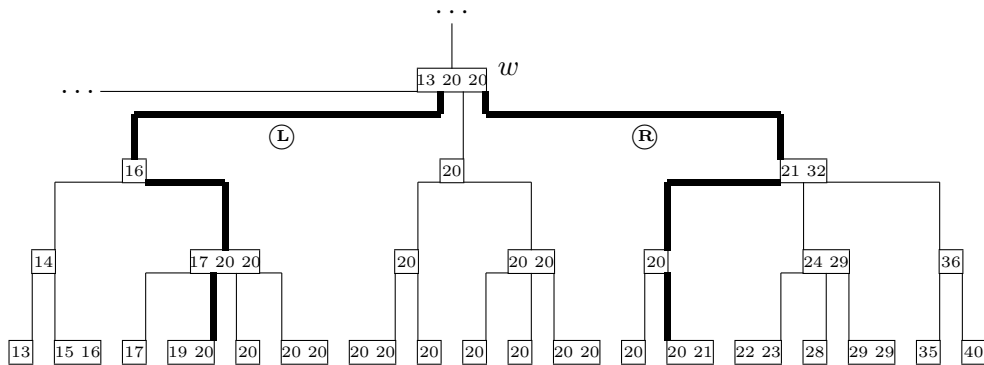


Figure 5: A portion of a (2,4)-Tree showing the locations of the entries with key=20. (Only keys are shown and leaves are not depicted.) Node  $w$  is the first node in the tree holding an entry with key=20. All entries to the right and to the left, respectively, of paths  $L$  and  $R$ , highlighted in bold, have key=20.

Based on the previous discussion, an immediate corollary of the above theorem is that the complexity of method `remove(e)` is

$$\Theta(\log n + s),$$

where  $s$  is the number of entries with key equal to `e.getKey()` in  $T$ .

Observe that the complexity of algorithm `EntrySearch` described for Binary Search Trees can be analyzed in a very similar way. In fact, `24EntrySearch` can be regarded as a generalization of that algorithm.

## References

[GT06] M.T. Goodrich and R. Tamassia. *Data Structures and Algorithms in Java*. John Wiley & Sons, New York NY, 4th edition, 2006.