Red-Black Trees

Akshal Aniche

February 20, 2018

This is the augmented transcript of a lecture given by Luc Devroye on the 13th of February 2018 for a Data Structures and Algorithms class (COMP 252) at McGill University about red-black trees.

Introduction

Since search trees are an extremely useful data structure, we want to design a balanced search tree, i.e. one with height $O(\log n)$.

A first attempt is to use a complete binary search tree. However, every time we would want to delete or insert a node, we would have to reconstruct the entire tree.

Question 1. *How can we design a practical balanced search tree?*

Such a tree would have a height of $O(\log n)$ and updates that would take $O(\log n)$ time, contrary to a complete tree.

The idea is to create a design with built-in "glue", so the tree is easily moldable.

Red-Black Trees

Definition 2. A **red-black tree** [1] is a binary search tree that has the following properties: [2]

- Leaves have no key. They are called **external nodes**. (Usually drawn as black squares).
- A node that is not a leaf has a key and is called an **internal node.**
- Each node has two additional properties, a *rank* and a *colour* (either red or black).
- A node is black and has rank r if and only if its parent has rank $r+1$.
- A node is red and has rank r if and only if its parent has rank r.
- All external nodes are black and have rank 0.
- If a node is red, none of its children or its parent can be red.

See tree 1 for an example of a valid red-black tree.

[1] Cormen et al. [2001]

[2] Red-black trees were invented by Bayer [1972].

For graph drawings of red-black trees, we will use the following notation:

● black internal node

🔴 red internal node

■ external node

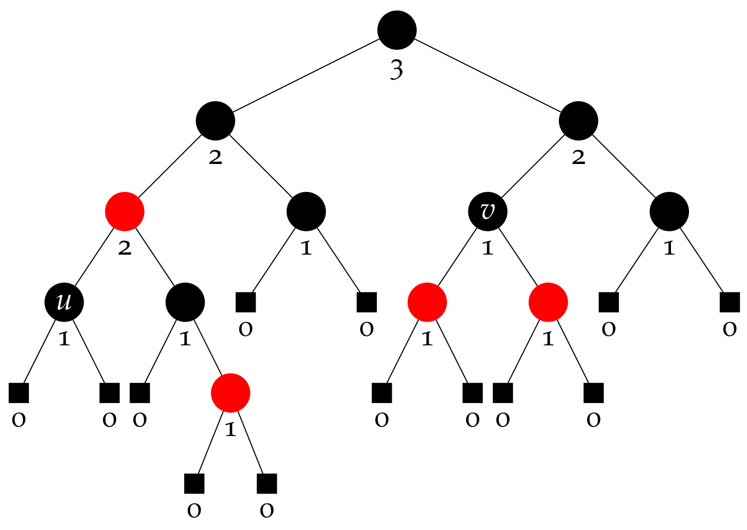


Figure 1: An example of a red-black tree (without explicit keys).
The rank of each node is written underneath the node.

Remark 3. Usually, we say that the root is black; the colour of the root doesn't affect the tree.

Remark 4. A binary tree can give multiple valid red-black trees. For instance, in the tree 1, the node v can also be coloured red, with rank 2. Then, its children would be black.

Remark 5. Even though all red-black trees are binary search trees, not every binary search tree is a valid red-black tree. For example, if the node denoted u was removed from tree 1, then the tree does not yield a valid red-black tree.

Definition 6. When implementing a red-black tree, we would create a cell with the following attributes:
-a key
-pointers to the left and right children and to the parent
-the color of the node
-the rank
We can deduce the color of every node if we have the ranks of all the nodes, and vice versa, so we can avoid either of them.

Proposition 7. *A red-black tree with n internal nodes has n+1 external nodes.*

2-3-4 tree view

We can think of ranks of black nodes as *levels*. We say that external nodes (rank=0) are at "sea level". Then, we can draw red-black trees with all the black nodes of the same rank at the same level. Red nodes are placed between levels.

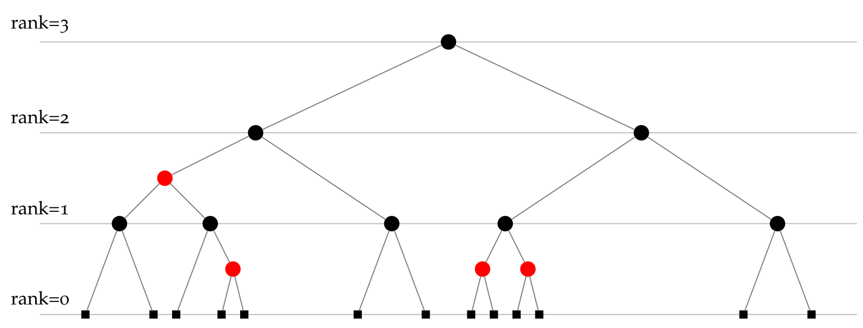


Figure 2: 2-3-4 tree view of tree 1

Definition 8. A black node and all its descendants until the next level of black nodes form a **pod**.

When red-black trees are represented this way, we can see three types of pods:

-pods with 2 black nodes in the lower level;

-pods with 3 black nodes in the lower level;

-pods with 4 black nodes in the lower level.

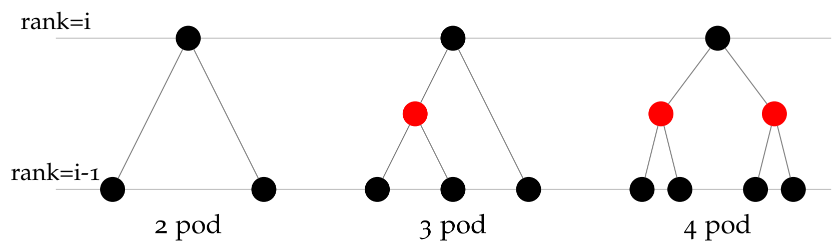


Figure 3: The tree types of pods in a Red-Black Tree

These three types of pods explain the name "2-3-4 tree".

Theorem 9. *Let r be the rank of the root of a red-black tree with n internal nodes. Then, $2^r \leq n+1 \leq 4^r$, or, equivalently, $r \leq \log_2(n+1) \leq 2r$.*

$n+1$ is the number of external nodes, which is the number of nodes on the 0^{th} level. Each node on level i had between two and four descendants on level $i-1$. Therefore, since, all the external nodes are descendants of the root, there are between 2^r and 4^r external nodes.

Exercise 10. Prove Theorem 9 using induction.

Claim 11. *For a red-black tree of height h with n internal nodes,*

$$h \leq 2\, rank[root] \leq 2\log_2(n+1)$$

This is justified by the fact that between two levels, you can have at most 2 nodes creating at most one additional layer, and there are rank[root] levels.

Operations

We will define the following operations for general red-black trees: **SEARCH**, **INSERT**, **DELETE**, **SPLIT**, and **JOIN**.

INSERT and DELETE

Definitions 12.

SEARCH is a standard search on a binary search tree.

INSERT is defined as a standard insert for binary search trees, followed by a **FIX** (see definition 14).

DELETE can be defined as a standard delete for binary search trees, followed by a **FIX** (see definition 14).

We can define a convenient operation, **LAZY DELETE**. It consists of marking the element that is supposed to be deleted, but leaving it in the data structure. Then, when more than 50% of the nodes have been marked, rebuild the tree.

To rebuild, get all unmarked nodes by inorder traversal in $O(n)$ time. Then, rebuild a red-black tree from the sorted list in $O(n)$ time.

Note that it is sufficient to make a complete binary tree and make all the nodes black except for the bottom level which should be red. Rebuilding the tree takes $O(n)$ time.

Exercise 13. Prove that a sequence of n **INSERT**/**DELETE**/**SEARCH** operations takes $O(n \log n)$ time.

We now turn to the **INSERT** operation.

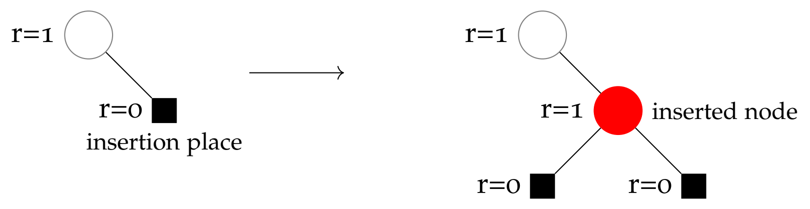


Figure 4: Illustration of standard insert on Red-Black Trees

From this picture, we can see that given the conditions on a valid red-black tree, the tree after insertion is invalid if the parent of the inserted node is also red. We will need to **FIX** the tree.

Definition 14.

We will use the notation : gp[x] for the grandparent of x; p[x] for the parent of x; u[x] for the sibling of the parent of x, also called the uncle of x.

FIX(x):

```
while $gp[x] \neq nil$ && $colour[p[x]] == red$ && $colour[u[x]] == red$
    // see Figure 5: Case 1
    $x = gp[x]$
    $rank[x] = rank[x] + 1$
    $colour[x] = red$
    $colour[left[x]] = black$
    $colour[right[x]] = black$
if $x == root$ then
    halt
else if $p[x] == root$ then
        if $colour[sibling[x]] == red$ then
        // see Figure 6: Case 2
            $rank[p[x]] = rank[p[x]] + 1$
            $colour[x] = black$
            $colour[sibling[x]] = black$
            halt
else if $colour[p[x]] == black$
    // Case 3
        halt
else // $colour[p[x]] == red$ && $colour[u[x]] == black$
    // see Figure 7: Case 4
    ROTATION // see Definition 15
    halt
```

Figure 5: Example of case 1

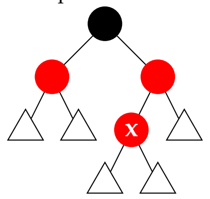
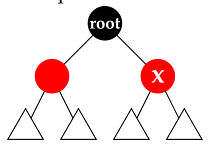


Figure 6: Example of case 2



Roughly half of the time, we end up with case 3, because a majority of the nodes in the tree are black by design.

Definition 15. ROTATION (Two cases, the other two cases are mirror symmetries)

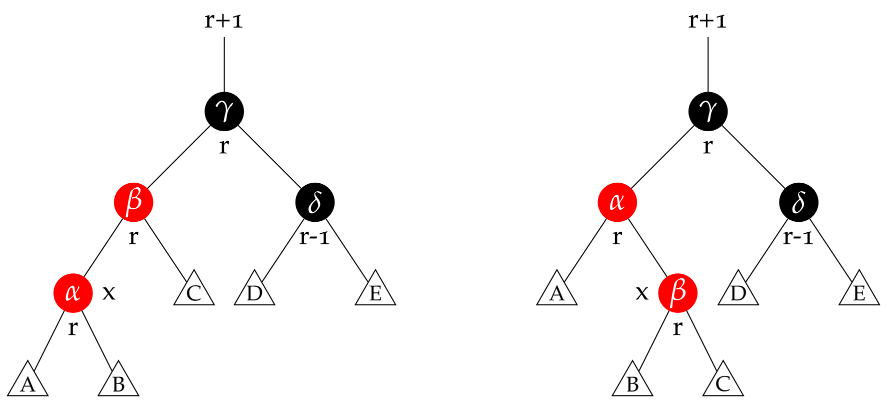


Figure 7: Two cases where we would need to effectuate a rotation (Case 4). The rank of each node is written underneath the node. $r + 1$ indicates the rank of the parent of the node γ

In both of these situations, we would get the following list if we traversed the subtree inorder: $A\alpha B\beta C\gamma D\delta E$. We want to change the structure of the tree so that we don't have two consecutive red nodes, but we want to preserve the order. For both configurations, we modify the subtree so that we have the same following tree 8:

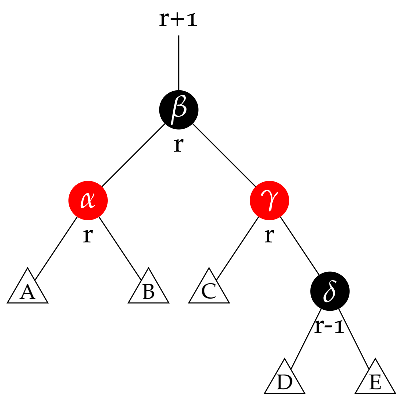


Figure 8: Result of the rotation of the two previous subtrees.
The parent of the node γ has become the parent of the node β.

Inorder traversal of the subtree rooted at β yields the desired list: $A\alpha B\beta C\gamma D\delta E$. For the other two cases, the picture is similar.

Observe that from the point of view of the root, of A, B, C, D, E, and the node of rank $r+1$, nothing has changed, because both before and after the operation, the ranks of adjacent nodes have not been altered.

ROTATION takes $O(1)$ time. In the **FIX** operation, the while loop accounts for $O(\log n)$ complexity, because the height of the tree is $O(\log n)$. Note that each **FIX** requires at most one **ROTATION**. This makes the red-black tree very efficient.

JOIN and SPLIT

Definition 16. **JOIN** takes in two red-black trees T_1, T_2 such that $\forall\ t_1 \in T_1, t_2 \in T_2,\ key[t_1] < key[t_2]$ and returns a red-black tree combining T_1 and T_2.

Definition 17. **SPLIT** takes a red-black tree T and a key k, and returns two red-black trees T_1 and T_2 where all the nodes in T_1 are the nodes in T with $key \leq k$ and all the nodes in T_2 are the nodes in T with $key > k$. (See example 22.)

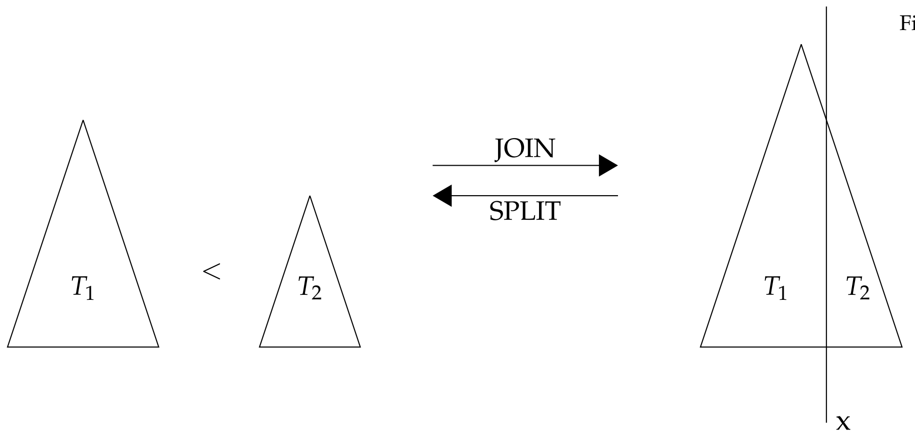


Figure 9: Illustration of JOIN and SPLIT

This illustrates how **JOIN** and **SPLIT** are supposed to work. The behaviour is similar to list concatenation and sublist. (This also means that we can implement lists by red-black trees, with the array index being the key.)

Now, we will write the **JOIN** operation. Assume $rank[root[T_2]] \leq rank[root[T_1]]$. (The other situation is similar.)

JOIN(T_1, T_2):

```
1   Find MIN($T_2$) (leftmost node); denote the node $j$.
2   Remove $j$ from $T_2$.
3   **FIX**($T_2 \backslash j$), and denote the resulting tree $B$, in $O(\log n)$ time.
4   Let $r$ be the rank of the root of $B$
5
6   Find in the right roof of $T_1$ a black node of rank $r$. $O(\log n)$
7   // The right roof of a tree is the rightmost path from the root.
8   Denote it $\alpha$. Let $\beta = p[\alpha]$, $A$ the subtree rooted at $\alpha$.
9   // see figure 10
10
11  $color[j] = red$
12  $rank[j] = r + 1$
13  $left[j] = \alpha$
14  $right[j] = root[B]$
15  $parent[j] = \beta$
16  Update the relevant pointers in the other nodes.
17  // Updating all these pointers takes time $O(1)$.
18  // see figure 11
19
20  **FIX**($j$) as in an ordinary red-black tree **INSERT**. ($O(\log n)$)
21  // see definition 14
```

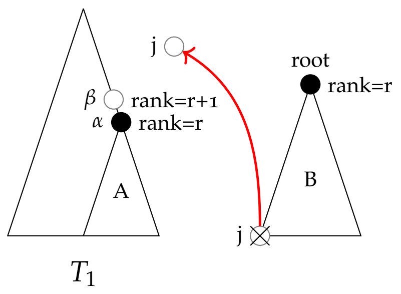


Figure 10: Step 1 and step 2

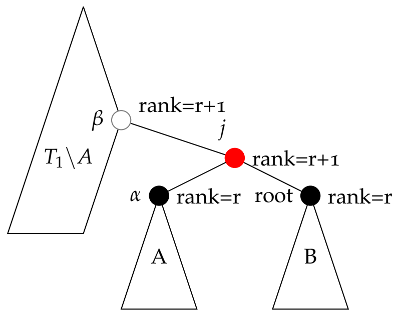


Figure 11: Step 3

Proposition 18. ***JOIN***(T_1, T_2) *takes* $O(\log n)$ *RAM time, where* n *is the total number of nodes in* T_1 *and* T_2 *combined.*

Remark 19. We can also write **JOIN**$(T_1$, x, $T_2)$, where all the nodes in T_1 are smaller than x and all the nodes in T_2 are larger than x. Then, let $j = x$, r be the rank of the root of T_2 and skip step 1.

Exercise 20. Formulate the algorithm for $rank[root[T_2]] \geq rank[root[T_1]]$.

We are now sketching **SPLIT**. We will show how to construct T_1. The construction of T_2 is symmetric.

SPLIT(T, k):

```
SEARCH($k$) to find the node with key $k$.
Keep track of the nodes with key smaller than or equal to $k$,
say $p_1, ..., p_i$, in decreasing order of key, and their respective
left subtree $A_1, A_2, ..., A_i$.
Let A be the subtree $A_1$ together with $p_1$.
// Note that $p_1$ has key $k$.

Repeatedly join the subtrees from right to left as such:
$S_2$ = JOIN($A, p_2, A_2$)
$S_3$ = JOIN($S_2, p_3, A_3$)
...
$S_{i-1}$ = JOIN($S_{i-2}, p_{i-1}, A_{i-1}$)
$S_i$ = JOIN($S_{i-1}, p_i, A_i$)

$T_1 = S_i$
return $T_1$
```

Exercise 21. Show that this takes $O(\log n)$ time overall. Hint: Show that a **JOIN** of two consecutive trees can be done in time $O(1)$, plus the difference in the rank of the roots.

Example 22.

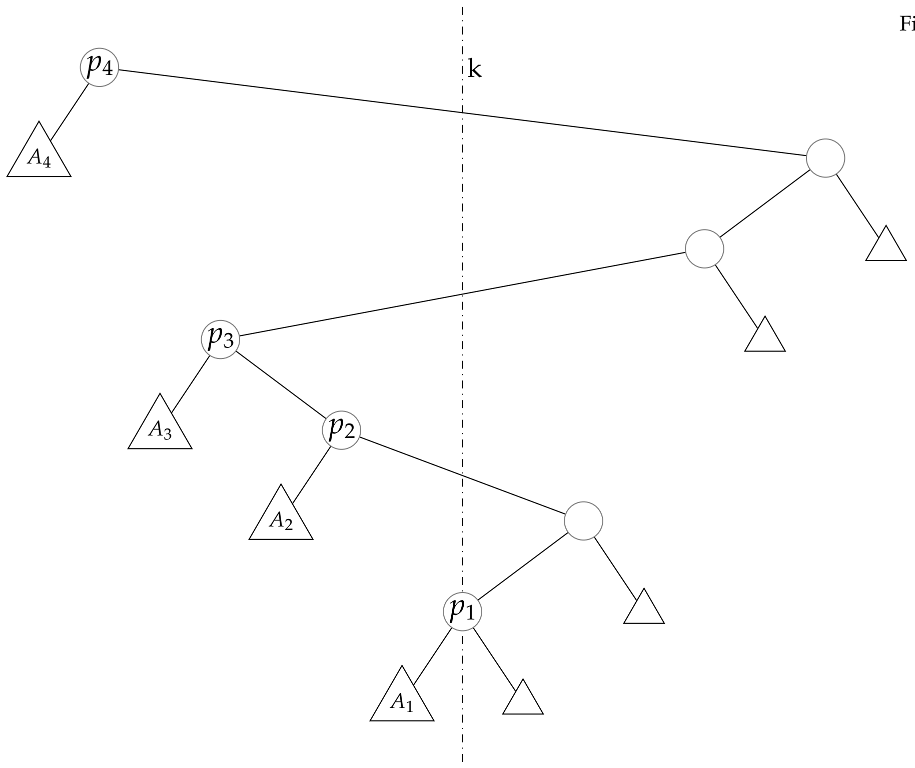
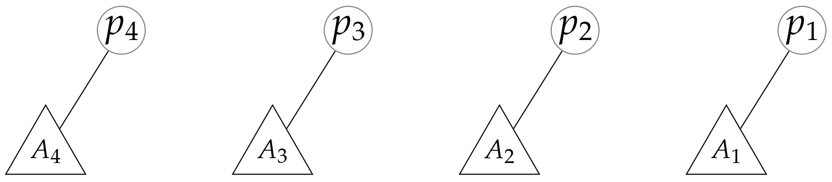


Figure 12: Step 1 of SPLIT

To construct T_1, the tree containing all the nodes of key$\leq k$, we must must **JOIN** the following subtrees.



This is done from right to left. Finally, return T_1 for the left split tree.

Exercise 23. Design an algorithm to construct T_2.

References

R. Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1(4):290–306, Dec 1972. DOI: 10.1007/BF00289509.

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2nd edition, 2001. ISBN 0-262-03293-7.