Red-Black Trees

Manolis Koubarakis
Red-Black Trees

• AVL trees and (2,4) trees have very nice properties, but:
  – AVL trees might need many rotations after a removal
  – (2,4) trees might require many split or fusion operations after an update

• **Red-black trees** are a data structure which requires only $O(1)$ structural changes after an update in order to remain balanced.
Definition

• A red-black tree is a binary search tree with nodes colored red and black in a way that satisfies the following properties:
  – **Root Property**: The root is black.
  – **External Property**: Every external node is black.
  – **Internal Property**: The children of a red node are black.
  – **Depth Property**: All the external nodes have the same black depth, defined as the number of black ancestors minus one (recall that a node is an ancestor of itself).
Example (2,4) Tree
In our figures, we use light blue color instead of black.
(2,4) Trees vs. Red-Black Trees

• Given a red-black tree, we can construct a corresponding (2,4) tree by merging every red node $v$ into its parent and storing the entry from $v$ at its parent.

• Given a (2,4) tree, we can transform it into a red-black tree by performing the following transformations for each internal node $v$:
  – If $v$ is a 2-node, then keep the (black) children of $v$ as is.
  – If $v$ is a 3-node, then create a new red node $w$, give $v$’s first two (black) children to $w$, and make $w$ and $v$’s third child be the two children of $v$ (the symmetric operation is also possible; see next slide).
    – If $v$ is a 4-node, then create two new red nodes $w$ and $z$, give $v$’s first two (black) children to $w$, give $v$’s last two (black) children to $z$, and make $w$ and $z$ be the two children of $v$.
(2,4) Trees vs. Red-Black Trees (cont’d)
Proposition

• The height of a red-black tree storing $n$ entries is $O(\log n)$.

• Proof?
Proof

• Let \( T \) be a red-black tree storing \( n \) entries, and let \( h \) be the height of \( T \). We will prove the following:

\[
\log(n + 1) \leq h \leq 2 \log(n + 1)
\]

• Let \( d \) be the common black depth of all the external nodes of \( T \). Let \( T' \) be the (2,4) tree associated with \( T \), and let \( h' \) be the height of \( T' \).

• Because of the correspondence between red-black trees and (2,4) trees, we know that \( h' = d \).

• Hence, \( d = h' \leq \log(n + 1) \) by the proposition for the height of (2,4) trees. By the internal node property of red-black trees, we have \( h \leq 2d \). Therefore, \( h \leq 2 \log(n + 1) \).
Proof (cont’d)

• The other inequality, $\log(n + 1) \leq h$ follows from the properties of proper binary trees and the fact that $T$ has $n$ internal nodes.
Updates

• Performing update operations in a red-black tree is similar to the operations of binary search trees, but we must additionally take care not to destroy the color properties.

• For an update operation in a red-black tree $T$, it is important to keep in mind the correspondence with a (2,4) tree $T'$ and the relevant update algorithms for (2,4) trees.
Insertion

• Let us consider the insertion of a new entry with key $k$ into a red-black tree $T$.
• We search for $k$ in $T$ until we reach an external node of $T$, and we replace this node with an internal node $z$, storing $(k, i)$ and having two external-node children.
• If $z$ is the root of $T$, we color $z$ black, else we color $z$ red. We also color the children of $z$ black.
• This operation corresponds to inserting $(k, i)$ into a node of the (2,4) tree $T'$ with external-node children.
• This operation preserves the root, external, and depth properties of $T$, but it might violate the internal property.
Insertion (cont’d)

• Indeed, if $z$ is not the root of $T$ and the parent $v$ of $z$ is red, then we have a parent and a child that are both red.

• In this case, by the root property, $v$ cannot be the root of $T$.

• By the internal property (which was previously satisfied), the parent $u$ of $v$ must be black.

• Since $z$ and its parent are red, but $z$’s grandparent $u$ is black, we call this violation of the internal property a double red at node $z$. 
Insertion (cont’d)

• To remedy a double red, we consider two cases.
• **Case 1: the sibling \( w \) of \( v \) is black.** In this case, the double red denotes the fact that we have created in our red-black tree \( T \) a **malformed** replacement for a corresponding 4-node of the (2,4) tree \( T' \), which has as its children the four black children of \( u, v \) and \( z \).
• Our malformed replacement has one red node (\( v \)) that is the parent of another red node (\( z \)) while we want it to have **two red nodes as siblings** instead.
• To fix this problem, we perform a **trinode restructuring** of \( T \) as follows.
Trinode Restructuring

• Take node $z$, its parent $v$, and grandparent $u$, and temporarily relabel them as $a$, $b$ and $c$, in left-to-right order, so that $a$, $b$ and $c$ will be visited in this order by an inorder tree traversal.

• Replace the grandparent $u$ with the node labeled $b$, and make nodes $a$ and $c$ the children of $b$ keeping inorder relationships unchanged.

• After restructuring, we color $b$ black and we color $a$ and $c$ red. Thus, the restructuring eliminates the double red problem.
Trinode Restructuring Graphically

Data Structures and Programming Techniques
Trinode Restructuring Graphically (cont’d)
Trinode Restructuring Graphically (cont’d)
Trinode Restructuring Graphically (cont’d)
Trinode Restructuring vs. Rotations

• The trinode restructuring operations are essentially the four kinds of rotations we discussed for AVL trees.
**Insertion (cont’d)**

- **Case 2: the sibling \( w \) of \( v \) is red.** In this case, the double red denotes an overflow in the corresponding (2,4) tree \( T' \). To fix the problem, we perform the equivalent of a split operation. Namely, we do a **recoloring**: we color \( v \) and \( w \) black and their parent \( u \) red (unless \( u \) is the root, in which case it is colored black).
- It is possible that, after such a recoloring, the double red problem **reappears** at \( u \) (if \( u \) has a red parent). Then, we repeat the consideration of the two cases.
- Thus, a recoloring either eliminates the double red problem at node \( z \) or propagates it to the grandparent \( u \) of \( z \).
- We continue going up \( T \) performing recoloring until we finally resolve the double red problem (either with a final recoloring or a trinode restructuring).
- Thus, the number of recolorings caused by insertion is no more than half the height of tree \( T \), that is, no more than \( \log(n + 1) \) by the previous proposition.
Recoloring

Data Structures and Programming Techniques
Recoloring (cont’d)
Example

• Let us now see some examples of insertions in an initially empty red-black tree.
Insert 4
Insert 7
Insert 12 – Double Red
After Restructuring
Insert 15 – Double Red

Data Structures and Programming Techniques
After Recoloring

Data Structures and Programming Techniques
Insert 3
Insert 5
Insert 14 – Double Red

Data Structures and Programming Techniques
Insertion of 18 – Double Red

Data Structures and Programming Techniques
After Recoloring

Data Structures and Programming Techniques
Insertion of 16 – Double Red
After Restructuring
Insertion of 17 – Double Red
After Recoloring – Double Red
After Restructuring

Data Structures and Programming Techniques
Proposition

• The insertion of a key-value entry in a red-black tree storing $n$ entries can be done in $O(\log n)$ time and requires $O(\log n)$ recolorings and one trinode restructuring.
Removal

- Let us now remove an entry with key $k$ from a red-black tree $T$.
- We proceed like in a binary tree search searching for a node $u$ storing such an entry.
- If $u$ does not have an external-node child, we find the internal node $v$ following $u$ in the inorder traversal of $T$. This node has an external-node child. We move the entry at $v$ to $u$, and perform the removal at $v$.
- Thus, we may consider only the removal of an entry with key $k$ stored at a node $v$ with an external-node child $w$. 
Removal (cont’d)

• To remove the entry with key $k$ from a node $v$ of $T$ with an external-node child $w$, we proceed as follows.
  
• Let $r$ be the sibling of $w$ and $x$ the parent of $v$. We remove nodes $v$ and $w$, and make $r$ a child of $x$.
  
• If $v$ was red (hence $r$ is black) or $r$ is red (hence $v$ was black), we color $r$ black and we are done.
Graphically
Removal (cont’d)

• If, instead, $r$ is black and $v$ is black, then, to preserve the depth property, we give $r$ a fictitious double black color.
• We now have a color violation, called the double black problem.
• A double black in $T$ denotes an underflow in the corresponding (2,4) tree $T’$.
• To remedy the double-black problem at $r$, we proceed as follows.
Removal (cont’d)

• Case 1: the sibling $y$ of $r$ is black and has a red child $z$.
• Resolving this case corresponds to a transfer operation in the (2,4) tree $T'$.
• We perform a trinode restructuring: we take the node $z$, its parent $y$, and grandparent $x$, we label them temporarily left to right as $a, b$ and $c$, and we replace $x$ with the node labeled $b$, making it parent of the other two nodes.
• We color $a$ and $c$ black, give $b$ the former color of $x$, and color $r$ black.
• This trinode restructuring eliminates the double black problem.
Restructuring a Red-Black Tree to Remedy the Double Black Problem

Data Structures and Programming Techniques
Restructuring (cont’d)
After the Restructuring
Removal (cont’d)

• Case 2: the sibling $y$ of $r$ is black and both children of $y$ are black.

• Resolving this case corresponds to a fusion operation in the corresponding (2,4) tree $T'$.

• We do a recoloring: we color $r$ black, we color $y$ red, and, if $x$ is red, we color it black; otherwise, we color $x$ double black.

• Hence, after this recoloring, the double black problem might reappear at the parent $x$ of $r$. We then repeat consideration of these three cases at $x$. 
Recoloring a Red-Black Tree that Fixes the Double Black Problem
After the Recoloring

Data Structures and Programming Techniques
Recoloring a Red-Black Tree that Propagates the Double Black Problem

Data Structures and Programming Techniques 54
After the Recoloring
Removal (cont’d)

• **Case 3**: the sibling \( y \) of \( r \) is red.

• In this case, we perform an **adjustment operation** as follows.

• If \( y \) is the right child of \( x \), let \( z \) be the right child of \( y \); otherwise, let \( z \) be the left child of \( y \).

• Execute the trinode restructuring operation which makes \( y \) the parent of \( x \).

• Color \( y \) black and \( x \) red.
Removal (cont’d)

• An adjustment corresponds to choosing a different representation of a 3-node in the (2,4) tree $T'$.

• After the adjustment operation, the sibling of $r$ is black, and either Case 1 or Case 2 applies, with a different meaning of $x$ and $y$.

• Note that if Case 2 applies, the double black problem cannot reappear.

• Thus, to complete Case 3 we make one more application of either Case 1 or Case 2 and we are done.

• Therefore, at most one adjustment is performed in a removal operation.
Adjustment of a Red-Black Tree in the Presence of a Double Black Problem
After the Adjustment
Removal (cont’d)

• The algorithm for removing an entry from a red-black tree with $n$ entries takes $O(\log n)$ time and performs $O(\log n)$ recolorings and at most one adjustment plus one additional trinode restructuring.
Example

• Let us now see a few removals from a given red-black tree.
Initial Tree
Remove 3
Remove 12 – Double Black
After Restructuring
Remove 17

Data Structures and Programming Techniques
Remove 18 – Double Black
After Recoloring
Remove 15
Remove 16 – Double Black
After the Adjustment – Double Black
After the Recoloring

Data Structures and Programming Techniques
Complexity of Operations in a Red-Black Tree

Time per level:
- Up phase: $O(1)$
- Down phase: $O(1)$
- Worst-case time: $O(\log n)$

Height:
- $O(\log n)$
Summary

• The red-black tree data structure is **slightly more complicated** than its corresponding (2,4) tree.

• However, the red-black tree has the conceptual advantage that only a **constant number of trinode restructurings** are ever needed to restore the balance after an update.
Readings

  – Section 10.5

• M. T. Goodrich, R. Tamassia. *Δομές Δεδομένων και Αλγόριθμοι σε Java. 5η έκδοση.* Εκδόσεις Δίαυλος.
  – Κεφ. 10.5

• R. Sedgewick. *Αλγόριθμοι σε C.*
  – Κεφ. 13.4