CMPSC250
Lecture 16: Red-Black Trees

Prof. John Wenskovitch
02/29/2016
Last Time

- Wrapping up Binary Search Trees
  - `delete()` method
- 2-3 Trees
  - Guarantee against BST worst case
  - Organizing and splitting nodes
2-3 Tree Analysis

• **Theorem:** Search and insert operations in a 2-3 Tree with \( N \) keys are guaranteed to visit at most \( \log(N) \) nodes.

• **Proof:**
  – Biggest height of a 2-3 Tree with \( N \) nodes is if all nodes are 2-Nodes.
    • In this case, height = \( \log(N) \).
  – Smallest height of a 2-3 Tree with \( N \) nodes is if all nodes are 3-Nodes.
    • In this case, height = \( \log_3(N) \).
  – Both of these fall in the \( O(\log(N)) \) complexity class.
2-3 Trees vs BSTs

• The height of a 2-3 Tree with 1 billion keys is guaranteed to be between 19-30.
• The height of a BST with 1 billion keys is guaranteed to be between 30-1,000,000,000.

• Downsides:
  – 2-3 Trees are certainly more complicated structures, and tougher to implement.
  – Because we need to maintain different types of nodes and rebuild the tree structure frequently, a 2-3 Tree may still be slower than a BST.

• 2-3 Trees provide insurance against the worst case.
2-3 Trees vs BSTs
We Did Better

<table>
<thead>
<tr>
<th>algorithm (data structure)</th>
<th>worst-case cost (after N inserts)</th>
<th>average-case cost (after N random inserts)</th>
<th>efficiently support ordered operations?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>search</td>
<td>insert</td>
<td>search hit</td>
</tr>
<tr>
<td>sequential search (unordered linked list)</td>
<td>N</td>
<td>N</td>
<td>N/2</td>
</tr>
<tr>
<td>binary search (ordered array)</td>
<td>lg N</td>
<td>N</td>
<td>lg N</td>
</tr>
<tr>
<td>binary tree search (BST)</td>
<td>N</td>
<td>N</td>
<td>1.39 lg N</td>
</tr>
<tr>
<td>2-3 tree search</td>
<td>2 lg N</td>
<td>2 lg N</td>
<td>1.00 lg N</td>
</tr>
</tbody>
</table>
Red-Black Trees

- A **Red-Black Tree** is a BST satisfying the following three restrictions:
  - Red links lean left.
  - No node has two red links connected to it.
  - The tree is **perfect black balance** – every path from the root to a null link has the same number of black links. We’ll call this number the tree’s **black height**.
Red-Black Trees

• This means that there is a one-to-one mapping between Red-Black BSTs and 2-3 Trees.
Handling Colors

- Since each Node is pointed to by precisely one link (from its parent), we encode the color of links in the Nodes by adding a boolean instance variable, setting red=true and black=false.

```java
private final boolean RED = true;
private final boolean BLACK = false;

Node {
    Key key;       // key
    Value val;     // linked data
    Node left, right;   // subtrees
    int N;               // # Nodes in subtree
    boolean color;      // color of link from parent
}  //Node
```
Color Issues

- This naïve implementation still allows for red links on right nodes, or two red links in a row.
- We need our tree to be self-correcting!
rotateLeft()

Node rotateLeft(Node h) {
    Node x = h.right;
    h.right = x.left;
    x.left = h;
    x.color = h.color;
    h.color = RED;
    x.N = h.N;
    h.N = 1 + size(h.left) + size(h.right);
    return x;
} //rotateLeft
Inserting into a Root 2-Node

- In a 2-3 Tree, we turned a single 2-node into a single 3-node to maintain tree balance. In a Red-Black Tree, this requires two different cases.
Inserting into a Non-Root 2-Node

• Same idea – but now we have to deal with a parent link if we rotate.
• Insert as if into a vanilla BST, then rotate if necessary to follow the Red-Black Tree rules.
Inserting into a Root 3-Node

• Now we have three cases to handle: the new key is larger, smaller, or in between the existing keys.

• Larger case:
Inserting into a Root 3-Node

• Smaller case:

• In-between case:
rotateRight()

Node rotateRight(Node h) {
    Node x = h.left;
    h.left = x.right;
    x.right = h;
    x.color = h.color;
    h.color = RED;
    x.N = h.N;
    h.N = 1 + size(h.left) + size(h.right);
    return x;
} //rotateRight
flipColors()

```java
void flipColors(Node h) {
    h.color = RED;
    h.left.color = BLACK;
    h.right.color = BLACK;
}
```

//flipColors
flipColors()

• A local transformation that preserves perfect black balance in the tree.

• Could color the root red.
  – Because a red node implies that it’s part of a 3-Node, we color the root black after every insert.

• Black height of the tree is increased by 1 every time the root is involved in a color flip.
Inserting into a 3-Node

• Same possible cases as inserting into an isolated 3-Node root.
  – The new link is connected to the right link of the 3-Node, in which case we flip the colors, or
  – The new link is connected to the left link of the 3-Node, in which case we call `rotateRight()` on the top link and then flip the colors, or
  – The new link is connected to the middle link of the 3-Node, in which case we call `rotateLeft()` on the bottom link, then call `rotateRight()` on the top link, and then flip the colors.
Inserting into a 3-Node
Passing a Red Link up the Tree

- From the parent’s point of view, a child link becoming red can be handled in the same way as if a RED link is caused by a direct new child that caused a 4-Node.
Red-Black Tree – Insertion

```java
void put(Key key, Value val) {
    root = put(root, key, val);
    root.color = BLACK;
} //put

Node put(Node h, Key key, Value val) {
    if (h == null) {
        return new Node(key, val, 1, RED);
    } //if
    int cmp = key.compareTo(h.key);
    if (cmp < 0) {
        h.left = put(h.left, key, val);
    } else if (cmp > 0) {
        h.right = put(h.right, key, val);
    } else {
        h.val = val;
    } //if-else
    if (isRed(h.right) && !isRed(h.left)) {
        h.rotateLeft(h);
    } //if
    if (isRed(h.left) && isRed(h.left.left)) {
        h.rotateRight(h);
    } //if
    if (isRed(h.left) && isRed(h.right)) {
        flipColors(h);
    } //if
    h.N = size(h.left) + size(h.right) + 1;
    return h;
} //put
```
Red-Black Tree – Insertion Visual

[Insertion process visualization diagram]
Red-Black Tree – Retrieval

Value get(Key key) {
    return get(root, key);
} //get

Value get(Node x, Key key) {
    if (x == null) {
        return null;
    } //if
    int cmp = key.compareTo(x.key);
    if (cmp < 0) {
        return get(x.left, key);
    } else if (cmp > 0) {
        return get(x.right, key);
    } else {
        return x.val;
    } //if-else
} //get
Red-Black Tree – Analysis

- Red-Black Trees are not necessarily perfectly balanced, but they’re close due to the correspondence with 2-3 Trees.
- **Theorem:** The height of a Red-Black Tree with $N$ nodes is no more than $\sim 2 \log(N)$.
- **Proof:**
  - The worst case height is a 2-3 Tree that is all 2-Nodes, except that the left-most path is made up of 3-Nodes.
  - Because each 3-Node includes an implicit RED link, processing each 3-Node takes 2 operations.
  - The path taking left links from the root is twice as long as the path of length $\log(N)$ that only involve 2-Nodes.
Red-Black Tree – Analysis

• The average performance increase from a Binary Search Tree to a Red-Black Tree is on the order of 40%.
Red-Black Tree – Analysis

[Graph showing cost and compares vs. operations with scale magnified by a factor of 250 compared to previous figures.]
### We Did Better

<table>
<thead>
<tr>
<th>Algorithm (data structure)</th>
<th>Worst-case cost (after N inserts)</th>
<th>Average-case cost (after N random inserts)</th>
<th>Efficiently support ordered operations?</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequential search</td>
<td>N</td>
<td>N/2</td>
<td>no</td>
</tr>
<tr>
<td>(unordered linked list)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>binary search</td>
<td>lg N</td>
<td>lg N</td>
<td>yes</td>
</tr>
<tr>
<td>(ordered array)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>binary tree search</td>
<td>N</td>
<td>1.39 lg N</td>
<td>yes</td>
</tr>
<tr>
<td>(BST)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2-3 tree search</td>
<td>2 lg N</td>
<td>1.00 lg N</td>
<td>yes</td>
</tr>
<tr>
<td>(red-black BST)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Red-Black Trees
How Do We delete()?

• Let’s consider an insertion algorithm for 2-3-4 Trees, in which the temporary 4-Nodes from 2-3 Trees can persist.
  – This insertion algorithm works in top-down fashion, and just wants to guarantee that the leaf node is not a 4-Node, so we have the ability to insert at the bottom.

• Cases:
  – If the root is a 4-Node, split it into three 2-Nodes and increase the height of the tree by 1.
  – If we find a 4-Node with a 2-Node parent, split the 4-Node into two 2-Nodes and pass the middle key up to make the parents a 3-Node.
How Do We delete()?

• More cases:
  – If we find a 4-Node with a 3-Node parent, split the 4-Node into two 2-Nodes and pass the middle key to the parent, making it a 4-Node.
    • (Because we handle the possibilities before they happen, we never need to worry about a 4-Node with a 4-Node parent.)
  • At the bottom of the tree, we are guaranteed to be inserting into either a 2-Node or a 3-Node.
How Do We delete()?

• How do we encode a 2-3-4 Tree as a Red-Black Tree?
  – 4-Nodes are a balanced subtree of three 2-Nodes, with both the left and right child connected to the parent with a RED link.
  – Split 3-Nodes on the way down the tree with color flips.
  – Balance 4-Nodes on the way up the tree with rotations, as for insertion.
How Do We delete()?

• How do we delete the minimum from a 2-3 subtree?
  – We can delete the key from a 3-Node easily and turn it into a 2-Node.
  – We cannot delete the key from a 2-Node without disrupting the balance of the tree.
• To guarantee that we don’t delete from a 2-Node, we perform transformations on the way down the tree. Let’s start with the root:
  – If the root is not a 2-Node, there is nothing to do.
  – If the root is a 2-Node and both children are 2-Nodes, we convert the 3-Nodes into a temporary 4-Node.
  – If the root is a 2-Node and has a 3-Node (or 4-Node) child, we borrow from the right child (if necessary) to guarantee that the left child of the root is not a 2-Node.
How Do We delete()?
How Do We delete()?

• On the way traversing down the tree, the current node is always a 2-Node or a 3-Node, and one of these cases must hold:
  – If the left child of the current node is not a 2-Node, there is nothing to do.
  – If the left child is a 2-Node and its immediate sibling is not a 2-Node, we move the smallest key from the sibling to the parent, and the smallest key from the parent to the left child.
  – If the left child and its immediate sibling are 2-Nodes, then we combine them with the smallest key in the parent to make a 4-Node, changing the parent from a 3-Node to a 2-Node or from a 4-Node to a 3-Node.

• In the end, the smallest key will either be in a 3-Node or a 4-Node. We can then split the temporary 4-Nodes on the way back up the tree.
How Do We `delete()`?
How Do We `delete()`?

• Now we can finally consider the general `delete()` case.
  – We still need to ensure that the current node is not a 2-Node during a search for the key to delete.
  – If we find the search key at the bottom of the tree, simply delete it.
  – If the key is not at the bottom, then we have to exchange it with its successor as in regular BSTs.
    • Then, since the current node is not a 2-Node, we have reduced the problem to deleting the minimum in a subtree whose root is not a 2-Node, and we can use the procedure for `deleteMin()`.
  – After deletion, split any 4-Nodes on the way back up the tree.
Any Questions?