Extra (optional) Material for Lecture 5 & 6

October 14&16, 2013

Red-Black Trees and More Amortized Analysis

October 17, 2013

Binary search trees are an elegant implementation of the dictionary data type, which requires support for

- `item SEARCH (item),`
- `void INSERT (item),`
- `void DELETE (item),`

and possible additional operations. Their main disadvantage is the worst case time $\Theta(n)$ for a single operation. The reasons are insertions and deletions that tend to get the tree unbalanced. It is possible to counteract this tendency with occasional local restructuring operations and to guarantee logarithmic time per operation.

**2-3-4 trees.** A special type of balanced tree is the 2-3-4 tree. Each internal node stores one, two, or three items and has two, three, or four children. Each leaf has the same depth. As shown in Figure 1, the items in the internal nodes separate the items stored in the subtrees and thus facilitate fast searching. In the smallest 2-3-4 tree of height $h$ every internal node has exactly two children, so we have $2^h$ leaves and $2^h - 1$ internal nodes. In the largest 2-3-4 tree of height $h$ every internal node has four children, so we have $4^h$ leaves and $(4^h - 1)/3$ internal nodes. We can store a 2-3-4 tree in a binary tree by expanding a node with $i > 1$ items and $i + 1$ children into $i$ nodes each with one item, as shown in Figure 2.

![Figure 1: A 2-3-4 tree of height two. All items are stored in internal nodes.](image1)

**Red-black trees.** Suppose we color each edge of a binary search tree either red or black. The color is conveniently stored in the lower node of the edge. Such an edge-colored tree is a red-black tree if

1. there are no two consecutive red edges on any descending path and every maximal such path ends with a black edge;
2. all maximal descending paths have the same number of black edges.

The number of black edges on a maximal descending path is the black height, denoted as $bh(\rho)$. When we transform a 2-3-4 tree into a binary tree as in Figure 2 we get a red-black tree. The result of transforming the tree in Figure 1 is shown in Figure 3.

![Figure 2: Transforming a 2-3-4 tree into a binary tree. Bold edges are called red and the others are called black.](image2)

![Figure 3: A red-black tree obtained from the 2-3-4 tree in Figure 1.](image3)

**Height Lemma.** A red-black tree with $n$ internal nodes has height at most $2 \log_2 (n + 1)$.

**Proof.** The number of leaves is $n + 1$. Contract each red edge to get a 2-3-4 tree with $n + 1$ leaves. Its height is $h \leq \log_2 (n + 1)$. We have $bh(\rho) = h$, and by Rule (1) the height of the red-black tree is at most $2bh(\rho) \leq 2 \log_2 (n + 1)$.

**Rotations.** Restructuring a red-black tree can be done with only one operation (and its symmetric version): a rotation that moves a subtree from one side to another, as shown in Figure 4. The inorder sequence of the left tree in Figure 4 is

$$\ldots, \text{inorder}(A), \nu, \text{inorder}(B), \mu, \text{inorder}(C), \ldots,$$

and this is also the inorder sequence of the right tree. In other words, a rotation maintains the inorder sequence. Function `ZIG` below implements the right rotation:

```
Node *ZIG(Node *µ)
assert µ ≠ NULL and ν = µ → ℓ ≠ NULL;
µ → ℓ = ν → r; ν → r = µ; return ν.
```
Function \( Z \) is symmetric and performs a left rotation. Occasionally it is necessary to perform two rotations in sequence, and it is convenient to combine them into a single operation referred to as a **double rotation**, as shown in Figure 5. We use a function \( Z_G Z_A \) to implement a double right rotation and the symmetric function \( Z_A Z_G \) to implement a double left rotation.

Node

Insertion. Before studying the details of the restructuring algorithms for red-black trees we look at the trees that arise in a short insertion sequence, as shown in Figure 6. After adding 10, 7, 13, 4 we have two red edges in sequence and repair this by promoting 10 (A). After adding 2 we repair the two red edges in sequence by a single rotation of 7 (B). After adding 5 we promote 4 (C), and after adding 6 we do a double rotation of 7 (D).

An item \( x \) is added by substituting a new internal node for a leaf at the appropriate position. To satisfy Rule (2) of the red-black tree definition, color the incoming edge of the new node red, as shown in Figure 7. Start the adjustment of color and structure at the parent \( \nu \) of the new node. We state the properties maintained by the insertion algorithm as invariants that apply to a node \( \nu \) traced by the algorithm.

**Invariant I.** The only possible violation of the red-black tree properties is that of Rule (1) at the node \( \nu \), and if \( \nu \) has a red incoming edge then it has exactly one red outgoing edge.
Case 2 has a symmetric case where left and right are interchanged. An insertion may cause logarithmically many promotions but at most two rotations.

**Deletion.** First find the node \( \pi \) that is to be removed. If necessary, we substitute the inorder successor for \( \pi \) or the inorder successor of the inorder successor so we can assume that both children of \( \pi \) are leaves. If \( \pi \) is last in inorder we substitute symmetrically. Replace \( \pi \) by a leaf \( \nu \), as shown in Figure 10. If the incoming edge of \( \pi \) is red then change it to black. Otherwise, remember the incoming edge of \( \nu \) as ‘double-black’, which counts as two black edges. Similar to insertions, it helps to understand the deletion algorithm in terms of a property it maintains.

**Invariant D.** The only possible violation of the red-black tree properties is a double-black incoming edge of \( \nu \).

Note that Invariant D holds right after we remove \( \pi \). We now present the analysis of all the possible cases. The adjustment operation is chosen depending on the local neighborhood of \( \nu \).

**Case 1.** The incoming edge of \( \nu \) is black. Done.

**Case 2.** The incoming edge of \( \nu \) is double-black. Let \( \mu \) be the parent and \( \kappa \) the sibling of \( \nu \). Assume \( \nu \) is left child of \( \mu \) and note that \( \kappa \) is internal.

**Case 2.1.** The edge from \( \mu \) to \( \kappa \) is black. As in Figure 11. Denote \( \mu \).

**Case 2.1.1.** Both outgoing edges of \( \kappa \) are black, as in Figure 11. Denote \( \mu \). Recurse for \( \nu = \mu \).

**Case 2.1.2.** The right outgoing edge of \( \kappa \) is red, as in Figure 12 to the left. Change the color of that edge to black and left rotate \( \mu \). Done.

**Case 2.1.3.** The right outgoing edge of \( \kappa \) is black, as in Figure 12 to the right. Change the color of the left outgoing edge to black and double left rotate \( \mu \). Done.

**Case 2.2.** The edge from \( \mu \) to \( \kappa \) is red, as in Figure 13. Left rotate \( \mu \). Recurse for \( \nu \).

Figure 9: Right rotation of \( \mu \) to the left and double right rotation of \( \mu \) to the right.

Figure 10: Deletion of node \( \pi \). The dashed edge counts as two black edges when we compute the black depth.

Figure 11: Demotion of \( \mu \).

Figure 12: Left rotation of \( \mu \) to the left and double left rotation of \( \mu \) to the right.

Figure 13: Left rotation of \( \mu \).

Case 2 has a symmetric case in which \( \nu \) is the right child of \( \mu \). Case 2.2 seems problematic because it recurses without moving \( \nu \) any closer to the root. However, the configuration excludes the possibility of Case 2.2 occurring again. If we enter Cases 2.1.2 or 2.1.3 then the termination is immediate. If we enter Case 2.1.1 then the termination follows because the incoming edge of \( \mu \) is red. The deletion may cause logarithmically many demotions but at most three rotations.

**Summary.** The red-black tree is an implementation of the dictionary data type and supports the operations search, insert, delete in logarithmic time each. An insertion or deletion requires the equivalent of at most three single rotations. The red-black tree also supports finding the minimum, maximum and the inorder successor, predecessor of a given node in logarithmic time each.

2-3-4 trees. As a more complicated application of amortization we consider 2-3-4 trees and the cost of restructuring them under insertions and deletions. We have seen 2-3-4 trees earlier when we talked about red-black trees. A set of keys is stored in sorted order the internal nodes of a 2-3-4 tree, which is characterized by the following rules:

1. each internal node has \( 2 \leq d \leq 4 \) children and stores \( d - 1 \) keys;
2. all leaves have the same depth.

As for binary trees, being sorted means that the inorder sequence of the keys is sorted. The only meaningful definition of the inorder sequence is the inorder sequence of the first subtree followed by the first key stored in the root followed by the inorder sequence of the second subtree followed by the second key, etc.
To insert a new key we attach a new leaf and add the key to the parent $\nu$ of that leaf. All is fine unless $\nu$ overflows because it now has five children. If it does we repair the violation of Rule (1) by climbing the tree one node at a time. We call an internal node non-saturated if it has fewer than four children.

Case 1. $\nu$ has five children and a non-saturated sibling to its left or right. Move one child from $\nu$ to that sibling, as in Figure 14.

![Figure 14](image)

Case 2. $\nu$ has five children and no non-saturated sibling. Split $\nu$ into two nodes and recurse for the parent of $\nu$, as in Figure 15. If $\nu$ has no parent then create a new root whose only children are the two nodes obtained from $\nu$.

![Figure 15](image)

Deleting a key is done in a similar fashion, although there we have to battle with nodes $\nu$ that have too few children rather than too many. Let $\nu$ have only one child. We repair Rule (1) by adopting a child from a sibling or by merging $\nu$ with a sibling. In the latter case the parent of $\nu$ looses a child and needs to be visited recursively. The two operations are illustrated in Figures 16 and 17.

![Figure 16](image)

Amortized analysis. The worst case for inserting a new key occurs when all internal nodes are saturated. The insertion then triggers logarithmically many splits. Symmetrically, the worst case for a deletion occurs when all internal nodes have only two children. The deletion then triggers logarithmically many mergers. Nevertheless we can show that in the amortized sense there are at most a constant number of split and merge operations per insertion and deletion.

We use the accounting method and store money in the internal nodes. The best internal nodes have three children because then they are flexible in both directions. They require no money, but all other nodes are given a positive amount to pay for future expenses caused by split and merge operations. Specifically, we store $\$4$, $\$1$, $\$0$, $\$3$, $\$6$ in each internal node with 1, 2, 3, 4, 5 children. As illustrated in Figures 14 and 16, an adoption moves money only from $\nu$ to its sibling. The operation keeps the total amount the same or decreases it, which is even better. As shown in Figure 15, a split frees up $\$5$ from $\nu$ and spends at most $\$3$ on the parent. The extra $\$2$ pay for the split operation. Similarly, a merger frees $\$5$ from the two affected nodes and at most $\$3$ on the parent. This is illustrated in Figure 17. An insertion makes an initial investment of at most $\$3$ to pay for creating a new leaf. Similarly, a deletion makes an initial investment of at most $\$3$ for destroying a leaf. This implies that for $n$ insertions and deletions we get at most $\frac{3n}{2}$ split and merge operations. In other words, the amortized number of split and merge operations is at most $\frac{3}{2}$.

Recall that there is a one-to-one correspondence between 2-3-4 tree and red-black trees. We can thus translate the above update procedure and get an algorithm for red-black trees with an amortized constant restructuring cost per insertion and deletion. We already proved that for red-black trees the number of rotations per insertion and deletion is at most a constant. The above argument implies that also the number of promotions and demotions is at most a constant, although in the amortized and not in the worst-case sense as for the rotations.