Splay Trees, II

This material is not covered in our textbook. You can read about splay trees in Section 7.3 of Data Structures and Their Algorithms by Lewis, Denenberg and about optimum weighted binary search trees in Section 6.2.2 of Sorting and Searching by Knuth.

Recall that we analyzed the amortized cost of splaying a node \( \nu \) in a binary search tree with root \( \varrho \) and found it to be at most \( 1 + 3[\beta(\varrho) - \beta(\nu)] \). We now use this result to show that splay trees have good amortized performance for all standard dictionary operations and more.

Access. To access an item \( b \), we first splay it to the root:

\[
\text{Node } * \text{ACCESS(Tree } \varrho, \text{ item } b) \\
\text{return SPLAY(} \varrho, b) \).
\]

We return the root even if it does not contain \( b \), one of the reasons being that we need to inform the calling program of the change in the tree. If \( b \) is not in the tree then the next smaller item, \( b^- \), or the next larger item, \( b^+ \), is splayed. The amortized cost is \( O(\beta(\varrho)) \).

Split. Given a item \( b \), we can split a splay tree into two, one containing all items \( x \leq b \) and the other all items \( x > b \), as illustrated in Figure 74. The amortized cost is the amortized cost for splaying plus the increase in the potential, which we denote as \( \Phi' - \Phi \). Recall that the potential of a collection of trees is the sum of the balances of all nodes. Splitting the tree decreases the number of descendents and therefore the balance of the root, which implies that \( \Phi' - \Phi < 0 \). It follows that the amortized cost of a split operation is less than that of a splay operation and therefore in \( O(\beta(\varrho)) \).

Join. Two splay trees can be merged into one if all items in one tree are smaller than all items in the other tree. The operation is illustrated in Figure 75. The cost for splaying the maximum item in the first tree is \( \Phi' - \Phi \). The potential increase caused by linking the two trees is

\[
\Phi' - \Phi \leq 2[\log_2 (s(\varrho_1) + s(\varrho_2))] - 2[\log_2 s(\varrho_1)] \\
\leq 2 + 2\log_2 \frac{s(\varrho_1) + s(\varrho_2)}{s(\varrho_1)} \\
\leq 2 + 2\log_2 (s(\varrho_2) + 1).
\]

The amortized cost of joining is therefore \( O(\beta(\varrho_1) + \beta(\varrho_2)) \).

Insert. To insert a new item, we split the tree. If \( b \) is already in the tree, we undo the split operation by linking the two trees. Otherwise, we make the two trees the left and right subtrees of a new node storing \( b \), as shown in Figure 76. The amortized cost for splaying is \( O(\beta(\varrho)) \). The potential increase caused by linking is

\[
\Phi' - \Phi \leq 2[\log_2 (s(\varrho_1) + s(\varrho_2) + 1)] \\
\leq \beta(\varrho) + 2.
\]
The amortized cost of an insertion is therefore $O(\beta(q))$.

Figure 76: There are three cases depending on whether $b$, $b^-$ or $b^+$ is splayed to the root.

**Delete.** To delete an item, we splay it to the root, remove the root, and join the two subtrees, as shown in Figure 77. Removing $b$ decreases the potential, and the amortized cost of joining the two subtrees is at most $O(\beta(q))$. This implies that the amortized cost of a deletion is at most $O(\beta(q))$.

![Figure 77: The item $b$ is in the tree then the root is removed and the two subtrees are joined to one tree.](image)

**Weighted Search.** A nice property of splay trees not shared by most other balanced trees is that they automatically adapt to biased search probabilities. The analysis is somewhat more involved and we only state the result. Each item or node has a positive weight, $w(\nu) > 0$, and we define $W = \sum_{\nu} w(\nu)$. We have the following generalization of the Investment Lemma, which we state without proof.

**Weighted Investment Lemma.** The amortized cost of splaying a node $\nu$ in a tree with total weight $W$ is $a \leq 1 + 3 \cdot \log_2 \frac{W}{w(\nu)}$.

It can be shown that this result is asymptotically best possible. To get a better feeling for what the result means, we study binary search trees that minimize the expected search time.

**Weighted Path Length.** Consider the problem of storing $n$ items $a_1 < a_2 < \ldots < a_n$ with probabilities $p_1, p_2, \ldots, p_n$ in a binary search tree $T$. To simplify the discussion, we only consider successful searches and thus assume $\sum_{i=1}^n p_i = 1$. The expected number of comparisons for a successful search is

$$1 + C(T) = \sum_{i=1}^n p_i \cdot (\delta_i + 1) = 1 + \sum_{i=1}^n p_i \cdot \delta_i,$$

where $\delta_i$ is the depth of the node that stores $a_i$. The quantity $C(T)$ is the weighted path length or the cost of $T$. We study the problem of constructing a tree that minimizes the cost. To develop an example, let $n = 3$ and $p_1 = \frac{1}{4}$, $p_2 = \frac{1}{3}$, $p_3 = \frac{1}{6}$. Figure 78 shows the five binary trees with three nodes and states their costs. We have seen earlier that the number of different binary trees with $n$ nodes is $\frac{1}{2}n(n-1)$. This is far too large to try all possibilities, unless $n$ is small, so we need to look for a more efficient way to construct an optimum tree.

![Figure 78: There are five different binary trees of three nodes. From left to right their costs are $\frac{1}{4}$, $\frac{3}{4}$, $\frac{1}{2}$, $\frac{5}{6}$, $\frac{7}{6}$. The first tree and the third tree are both optimal.](image)

**Dynamic Programming.** We write $T_i^j$ for the optimum weighted binary search tree of $a_i, a_{i+1}, \ldots, a_j$, $C_i^j$ for its cost, and $p_i^j = \sum_{k=i}^j p_k$ for the total probability of the items in $T_i^j$. Suppose we know that the optimum tree stores item $a_k$ in its root. Then the left subtree is $T_i^k-1$ and the right subtree is $T_{k+1}^j$. The cost of the optimum tree is therefore $C_i^j = C_i^{k-1} + C_{k+1}^j + p_i^{k-1} + p_i^j$. We note that $p_i^{k-1} + p_i^j = p_i^j - p_k$. Since we do not now which item is in the root, we try all possibilities and find the minimum:

$$C_i^j = \min_{i \leq k \leq j} \{C_i^{k-1} + C_{k+1}^j + p_i^j - p_k\}.$$
This formula can be translated directly into a dynamic programming algorithm. We use three two-dimensional arrays, one for the sums of probabilities $p_i^j$, one for the costs $C_i^j$ of optimum trees, and one for the indices $R_i^j$ of the items stored in their roots. We assume that the first array has already been computed.

```
float OPT
    for k = 1 to n do
        C[k, k] = 0; R[k, k] = k
    endfor;
    for ℓ = 2 to n do
        j = i + ℓ - 1; C[i, j] = ∞;
        for k = i to j do
            cost = C[i, k - 1] + C[k + 1, j] + p[i, j] - p[k, k];
            if cost < C[i, j] then
                C[i, j] = cost; R[i, j] = k
            endif
        endfor
    endfor
    return C[1, n].
```

To correctly compute the cost also for the boundary cases $k = i, j$, we need to define $C[x, y] = 0$ whenever $y < x$. This is easiest done by adding another row at the end and another column at the beginning of the array. The main part of the algorithm consists of three nested loops each iterating through at most $n$ values. The running time is therefore in $O(n^3)$.

**Example.** Table 5 shows the partial sums of probabilities for the data in the earlier example. Table 6 shows the costs and the indices of the root items of the optimum trees computed for all contiguous subsequences of items. The optimum tree can be constructed from $R$ as follows. The root stores the item with index $R[1, 3] = 1$. The left subtree is therefore empty and the right subtree stores $a_2, a_3$. The root of the optimum right subtree stores the item with index $R[2, 3] = 2$. Again the left subtree is empty and the right subtree consists of a single node storing $a_3$.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Table 5: Six times the partial sums of probabilities used by the dynamic programming algorithm.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

Table 6: Six times the costs and the roots of the optimum trees.

**Improved running time.** Notice that the array $R$ in Table 6 is monotonically non-decreasing, both along rows and along columns. Indeed it is possible to prove $R_i^{j-1} ≤ R_i^j$ in every row and $R_i^j ≤ R_{i+1}^j$ in every column. We omit the proof of these inequalities and instead show how they can be used to improve the dynamic programming algorithm. Instead of trying all roots from $i$ through $j$ we can restrict the innermost for-loop to

```
for k = R[i, j - 1] to R[i + 1, j] do
```

The monotonicity property implies that this change does not alter the result of the algorithm. The running time of a single iteration of the middle for-loop is now

$$\sum_{i=1}^{n-ℓ+1} (R_i^{j+1} - R_i^{j+1} + 1) \leq n + R_{n-ℓ+2}^n - R_1^{j+1} \leq 2n$$

because $j = i + ℓ - 1$ and the highest index of any item is $n$. In words, each iteration of the middle for-loop takes only time $O(n)$, which implies that the entire algorithm takes only time $O(n^2)$.