Binary Search Trees

One of the purposes of sorting is to facilitate fast searching. However, while a sorted sequence stored in a linear array is good for searching, it is expensive to add and delete items. Binary search trees give you the best of both worlds: fast search and fast update.

Definitions and terminology. We begin with a recursive definition of the most common type of tree used in algorithms. A *(rooted) binary tree* is either empty or a node (the *root*) with a binary tree as left subtree and binary tree as right subtree. We store items in the nodes of the tree. It is often convenient to say the items *are* the nodes. A binary tree is sorted if each item is between the smaller or equal items in the left subtree and the larger or equal items in the right subtree. For example, the tree illustrated in Figure 11 is sorted assuming the usual ordering of English characters. Terms for relations between family members such as *child*, *parent*, *sibling* are also used for nodes in a tree. Every node has one parent, except the root which has no parent. A *leaf* or *external node* is one without children; all other nodes are *internal*. A node ν is a *descendent* of μ if $\nu = \mu$ or ν is a descendent of a child of μ. Symmetrically, μ is an *ancestor* of ν if ν is a descendent of μ. The *subtree* of μ consists of all descendents of μ. An *edge* is a parent-child pair.

The *size* of the tree is the number of nodes. A binary tree is *full* if every internal node has two children. Every full binary tree has one more leaf than internal node. To count its edges we can either count 2 for each internal node or 1 for every node other than the root. Either way, the total number of edges is one less than the size of the tree. A *path* is a sequence of contiguous edges without repetitions. Usually we only consider paths that descend or paths that ascend. The *length* of a path is the number of edges. For every node μ there is a unique path from the root to μ. The length of that path is the *depth* of μ. The *height* of the tree is the maximum depth of any node. The

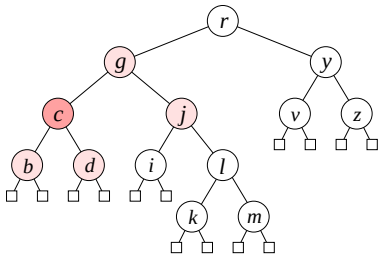

Figure 11: The parent, sibling and two children of the dark node are shaded. The internal nodes are drawn as circles while the leaves are drawn as squares.

path length is the sum of depths over all nodes, and the *external path length* is the same sum restricted to the leaves in the tree.

Searching. A *binary search tree* is a sorted binary tree. We assume each node is a record storing an item and pointers to two children:

```
struct Node {item info; Node *ℓ, *r};
typedef Node *Tree.
```

Sometimes it is convenient to also store a pointer to the parent, but for now we will do without. We can search in a binary search tree by tracing a path starting at the root.

```
Node *SEARCH(Tree ϱ, item x)
  case ϱ = NULL: return NULL;
       x < ϱ → info: return SEARCH(ϱ → ℓ, x);
       x = ϱ → info: return ϱ;
       x > ϱ → info: return SEARCH(ϱ → r, x)
  endcase.
```

The running time depends on the length of the path, which is at most the height of the tree. Let n be the size. In the

worst case the tree is a linked list and searching takes time $O(n)$. In the best case the tree is perfectly balanced and searching takes only time $O(\log n)$.

Insert. To add a new item is similarly straightforward: follow a path from the root to a leaf and replace that leaf by a new node storing the item. Figure 12 shows the tree obtained after adding w to the tree in Figure 11. Again

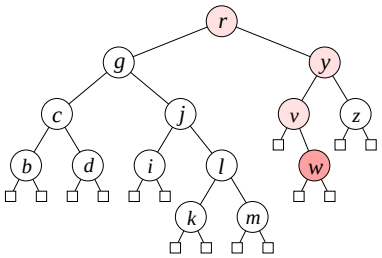

Figure 12: The shaded nodes indicate the path from the root we traverse when we insert w into the sorted tree.

the running time depends on the length of the path. If the insertions come in a random order then the tree is usually close to being perfectly balanced. Indeed, the tree is the same as the one that arises in the analysis of quicksort. The expected number of comparisons for a (successful) search is one n-th of the expected running time of quicksort, which is roughly $2 \ln n$.

Delete. The main idea for deleting an item is the same as for inserting: follow the path from the root to the node ν that stores the item.

`Case 1.` ν has no internal node as a child. Remove ν.

`Case 2.` ν has one internal child. Make that child the child of the parent of ν.

`Case 3.` ν has two internal children. Find the rightmost internal node in the left subtree, remove it, and substitute it for ν, as shown in Figure 13.

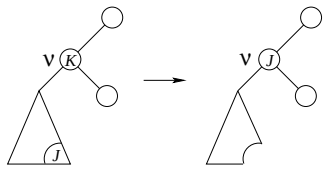

Figure 13: Store J in ν and delete the node that used to store J.

The analysis of the expected search time in a binary search tree constructed by a random sequence of insertions and deletions is considerably more challenging than if no deletions are present. Even the definition of a random sequence is ambiguous in this case.

Optimal binary search trees. Instead of hoping the incremental construction yields a shallow tree, we can construct the tree that minimizes the search time. We consider the common problem in which items have different probabilities to be the target of a search. For example, some words in the English dictionary are more commonly searched than others and are therefore assigned a higher probability. Let $a_1 < a_2 < \ldots < a_n$ be the items and p_i the corresponding probabilities. 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 in a binary search tree T storing the n items is

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

where δ_i is the depth of the node that stores a_i. $C(T)$ 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 $p_1 = \frac{1}{2}$, $p_2 = \frac{1}{3}$, $p_3 = \frac{1}{6}$. Figure 14 shows the five binary trees with three nodes and states their costs. It can be shown that the

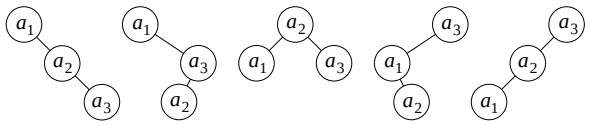

Figure 14: There are five different binary trees of three nodes. From left to right their costs are $\frac{2}{3}, \frac{5}{6}, \frac{2}{3}, \frac{7}{6}, \frac{4}{3}$. The first tree and the third tree are both optimal.

number of different binary trees with n nodes is $\frac{1}{n+1}\binom{2n}{n}$, which is exponential in n. This is far too large to try all possibilities, so we need to look for a more efficient way to construct an optimum tree.

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 $C_i^j = C_i^{k-1} + C_{k+1}^j + p_i^j - p_k$. Since we do not know which item is in the root we try all possibilities and find the minimum:

$$C_i^j \quad = \quad \min_{i \le k \le j} \{C_i^{k-1} + C_{k+1}^j + p_i^j - p_k\}.$$

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 of optimum trees, C_i^j, and one for the indices of the items stored in their roots, R_i^j. We assume that the first array has already been computed. We initialize the other two arrays along the main diagonal and add one dummy diagonal for the cost.

```
for k = 1 to n do
  C[k, k-1] = C[k, k] = 0;  R[k, k] = k
endfor ;
C[n+1, n] = 0.
```

We fill the rest of the two arrays one diagonal at a time.

```
for ℓ = 2 to n do
  for i = 1 to n − ℓ + 1 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
endfor .
```

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 1 shows the partial sums of probabilities for the data in the earlier example. Table 2 shows the costs and the indices of the roots of the optimum trees computed for all contiguous subsequences. 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.

$6p$	1	2	3
1	3	5	6
2		2	3
3			1

Table 1: Six times the partial sums of probabilities used by the dynamic programming algorithm.

$6C$	1	2	3
1	0	2	4
2		0	1
3			0

R	1	2	3
1	1	1	1
2		2	2
3			3

Table 2: Six times the costs and the roots of the optimum trees.

Improved running time. Notice that the array R in Table 2 is monotonic, both along rows and along columns. Indeed it is possible to prove $R_i^{j-1} \le R_i^j$ in every row and $R_i^j \le R_{i+1}^j$ in every column. We omit the proof and show how the two inequalities can be used to improve the dynamic programming algorithm. Instead of trying all roots from i through j we restrict the innermost `for` -loop to

$$\texttt{for } k = R[i, j-1] \texttt{ to } R[i+1, j] \texttt{ do}$$

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

$$U_\ell(n) \quad = \quad \sum_{i=1}^{n-\ell+1} (R_{i+1}^j - R_i^{j-1} + 1).$$

Recall that $j = i + \ell - 1$ and note that most terms cancel, giving

$$\begin{aligned} U_\ell(n) \quad &= \quad R_{n-\ell+2}^n - R_1^{\ell-1} + (n - \ell + 1) \\ &\le \quad 2n. \end{aligned}$$

In words, each iteration of the outer `for` -loop takes only time $O(n)$, which implies that the entire algorithm takes only time $O(n^2)$.