Chapter 4

Fractional Cascading

In this chapter, we study an algorithm design principle called *fractional cascading.* Essentially, fractional cascading says that many problems can be solved by first solving (recursively) a subproblem whose size is a constant fraction of the original problem size and then using this solution to get back to a solution of the original problem.

4.1 Iterated Search

Consider the following *iterated search* problem. We are given $h < n$ sorted arrays of numbers $A^{(1)}, \ldots, A^{(h)}$ each of length n. We want to preprocess these arrays so that we can quickly locate a query key k in each of the h arrays. That is, for each $1 \leq i \leq h$ we want to know the smallest value in $A^{(i)}$ greater than or equal to k. For convenience, so that the output is well-defined, we define $A^{(i)}_\infty = \infty$.

Since the arrays are sorted, we can solve this problem without any preprocessing by using binary search to locate k in each of the arrays. Since there are h arrays and binary search takes $O(\log n)$ time, the query time we get from this is $O(h \log n)$. Can we do better?

Suppose we merge all the arrays into a single sorted array A' as in Figure 4.1. Each element in A' keeps track of which array $A^{(i)}$ that it came from. With each element x in A' we also associate h integers $x_1, \ldots, x_h$, where x_i is smallest integer such that $A'_{x_i} \geq x$ and A'_{x_i} came from array $A^{(i)}$. Once this data structure is built, answering a query takes $O(h + \log hn) = O(h + \log n)$ time, since we only have to do binary search on A' to find the element x and then report $A'_{x_1}, \ldots, A'_{x_h}$.

A query time of $O(h + \log n)$ is certainly better than the $O(h \log n)$ query time we got without building the data structure, but what price did we pay for this query time? The array A' contains hn entries, and each entry contains h integers $(x_1, \ldots, x_h)$, so this array takes up a memory of size $\Theta(h^2 n)$. Constructing the array can be done fairly easily in $O(hn \log n + h^2 n)$ time by first sorting (in $O(hn \log n)$ time) and then scanning the sorted array using an auxilliary array of size h (in $O(h^2 n)$ time) to compute the x_i values for each array entry.

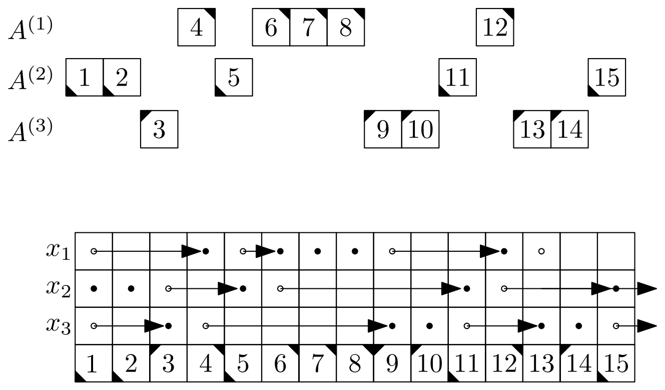

Figure 4.1: A first solution to the iterated search problem obtained by merging the $h = 3$ input arrays of size $n = 5$ into one sorted array of size $hn = 15$.

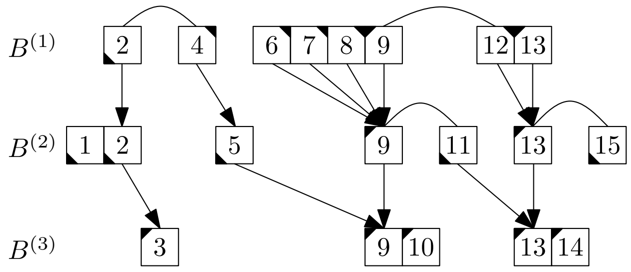

Figure 4.2: An iterated search data structure using fractional cascading. Loops in right pointers are ommitted to reduce clutter.

The $O(h^2 n)$ term in the running time and memory requirements seems unfortunate. Luckily, it can be avoided by using fractional cascading. Let us define $B^{(h)} = A^{(h)}$ and suppose we take a sample of $B^{(h)}$ by selecting every second element, starting with the second element. In this way, we obtain a sample $S^{(h)}$ of size $\lfloor n/2 \rfloor$. Next, we merge $S^{(h)}$ with A^{h-1} to get a sorted array $B^{(h-1)}$. For each element x in $B^{(h-1)}$ we maintain two extra values, down(x) and right(x). The value of down(x) is the smallest integer i such that $B^{(h)}_i \geq x$. If x comes from $A^{(h-1)}$, then the integer right(x) is the index of x, otherwise right(x) is the index of the first element of $B^{(h-1)}$ that comes from $A^{(h-1)}$ and appears after x. See Figure 4.2 for an example.

Above, we've outlined a procedure that takes as input $B^{(h)}$ and $A^{(h-1)}$ and output $B^{(h-1)}$. We can now repeat this procedure, using $A^{(h-2)}$ and $B^{(h-1)}$ to obtain $B^{(h-2)}$, and so on until we get $B^{(1)}$. We claim that this data structure can be used to solve the iterated search problem in $O(h + \log n)$ time.

Suppose we have done something to $B^{(i)}$, $1 \leq i < h$ to find the smallest integer i such that $B^{(1)}_i \geq k$. Let x denote $B^{(1)}_i$. Then the smallest element of $A^{(i)}$ greater than k is $A^{(i)}_{\mathtt{right}(x)}$, so locating k in $B^{(i)}$ is sufficient to locate k in $A^{(i)}$. Furthermore, we can locate k in $B^{(i+1)}$ by looking at down(x). It is not hard to verify that the smallest index j such that $B^{(i+1)}_j \geq k$ is either down(x) or down$(x) - 1$. (It can't be down$(x) - 2$ otherwise x would not be the smallest element in $B^{(i)}$ larger than x.)

This is enough to solve an iterated search query because we can first locate the query key k in $B^{(1)}$ using binary search and the repeatedly use the location of k in $B^{(i)}$ to locate k in $A^{(i)}$ and $B^{(i+1)}$, for each $1 \leq i < h$. The binary search takes $O(\log n)$ time and each subsequent step takes constant time, for a total of $O(h + \log n)$.

Next we analyze the preprocessing and storage requirements for this data structure. Let $|X|$ denote the size of the array X. Since $B^{(i)}$ is obtained by sampling every second element from $B^{(i+1)}$ and merging this sample with $A^{(i)}$, we have

$$\begin{aligned}
|B^{(h)}| &= n \\
|B^{(h-1)}| &\leq n + \frac{1}{2}|B^{(h)}| \leq n + \frac{1}{2}n \\
|B^{(h-2)}| &\leq n + \frac{1}{2}|B^{(h-1)}| \leq n + \frac{1}{2}n + \frac{1}{4}n \\
|B^{(h-3)}| &\leq n + \frac{1}{2}|B^{(h-2)}| \leq n + \frac{1}{2}n + \frac{1}{4}n + \frac{1}{8}n
\end{aligned}$$

and, in general,

$$|B^{(i)}| \leq \sum_{j=0}^{\infty} n/2^j = 2n$$

Therefore, the total number of array entries in the entire data structure is $O(hn)$, and since each array entry contains only a constant amount of data, the entire data structure has size $O(hn)$. It is not hard to implement the preprocessing so that computing $B^{(i)}$ takes $O(n)$ time, so the preprocessing and memory requirements are $O(hn)$.

Theorem 8. *Given $h \leq n$ sorted arrays each of size n, there exists a data structure requiring $O(hn)$ preprocessing time and memory that answers iterated search queries in $O(h + \log n)$ time.*

4.2 Segment Trees

To be done later.

4.3 Skip Lists

In the last section, we saw how fractional cascading can help when we are trying to locate the same element in many sorted arrays. In this section, we will see fractional cascading can be used to locate an element in 1 sorted array.

Let $X = \{k_1, \ldots, k_n\}$ be a set of n distinct real numbers, where $k_i < k_{i+1}$ for all $1 \leq i < n$. For convenience, we also define $k_0 = -\infty$ and $k_{n+1} = +\infty$. We define a *gradation* $X^{(0)}, \ldots, X^{(h)}$ as follows. The level $X^{(0)} = X$. The *level* $X^{(i)}$, $i > 0$, is obtained by selecting each element from $X^{(i-1)}$ with a fixed constant probability $0 < p < 1$. The value h is called the *height* of the skip list and is defined as $h = \min\{i : |X^{(i+1)}| = 0\}$.

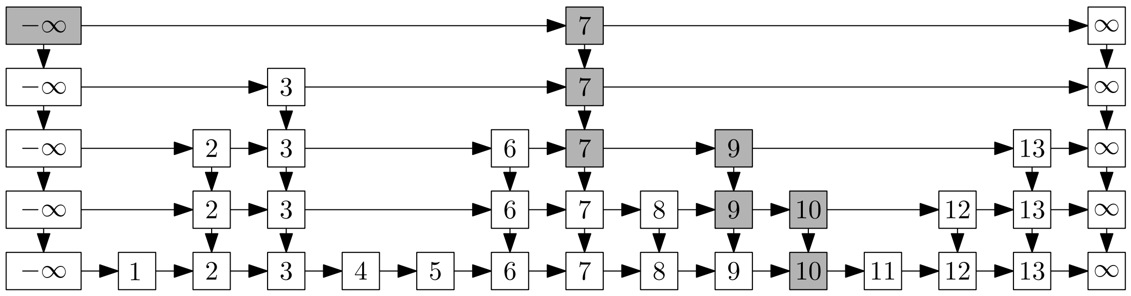

Figure 4.3: A skip list. The grey nodes show the search path for the key 11.

In a *skip list*, we maintain each level in a sorted list. More precisely, for each level $X^{(i)}$, we maintain $\{k_0, k_{n+1}\} \cup X^{(i)}$ in a sorted list L_i. We use the notation $\mathrm{right}(v)$ to denote the successor of a node v in one of these lists. Each element of L_i, $i > 0$ also maintains another pointer $\mathrm{down}(x)$ which points to the same element in L_{i+1}. Note that k_0 and k_{n+1} appear as the first and last elements, respectively, of every list. We call the first node of L_h the *root* of the skip list. See Figure 4.3 for an example of a skip list generated by coin tosses.

To search for the key k in a skip list, we start by setting v equal to the root of the skip list and repeating the following step: If the value stored at $\mathrm{right}(v)$ is less than q then we set v equal to $\mathrm{right}(v)$, otherwise we set $v = \mathrm{down}(v)$. The search ends when we try to set $v = \mathrm{down}(v)$ but $\mathrm{down}(v)$ is undefined. It is not hard to verify that the search ends at a node v whose value is the largest value k_i smaller than k. (This follows from the invariant that the value stored at v is always less than k and the search terminates at level 0.) The highlighted path in Figure 4.3 shows the search path for key 11.

To insert a new key k into a skip list we first choose a random height l for k, according to the distribution $\Pr\{l = i\} = p^i(1-p)$. We then follow the search path of k in the skip list, except any time we follow a down pointer at level $i \leq l$ we splice a new node into the skip list at level i whose value is k. If the value $h = \lceil \log_{1/p} n \rceil$ changes due to the insertion of the new element, then we add one more level to the skiplist and add each node at level $h-1$ to this new level with probability p.

To delete the key k from a skip list, we follow the search path for k in the list, except that any time the data at $\mathrm{right}(v)$ is equal to k we splice $\mathrm{right}(v)$ out of the list. If the value $k = \lceil \log_{1/p} n \rceil$ decreases due to the deletion of k then we remove the top level of the skip list.

Analysis. To start with something simple, we first analyze the expected number of nodes at level i. A value k appears at level i because it was selected in the first i gradation steps. The probability that this happens is p^i. Therefore, by linearity of expectation, the expected number of nodes at level i is $\mathbf{E}[|L_i|] \leq 2 + np^i$. (The extra 2 counts the elements k_0 and k_{n+1}, which appear on every level.) This works fine for small values of i, but when i gets big the extra 2 nodes start to add up. To resolve this, we note that, for any level, i, $|L_i| \leq 3|X^{(i)}|$. Taking expected values on both sides of this equation, we see that $\mathbf{E}[|L_i|] \leq 3np^i$.

We can now analyze the expected number of nodes in the skiplist as

$$\begin{aligned}
\mathbf{E}\left[\sum_{i=0}^{\infty}|L_i|\right] &= \sum_{i=0}^{\infty}\mathbf{E}[|L_i|] \\
&= \sum_{i=0}^{\lceil\log_{1/p} n\rceil}\mathbf{E}[|L_i|] + \sum_{i=\lceil\log_{1/p} n\rceil+1}^{\infty}\mathbf{E}[|L_i|] \\
&\le \sum_{i=0}^{\lceil\log_{1/p} n\rceil}(2+np^i) + \sum_{i=\lceil\log_{1/p} n\rceil+1}^{\infty}3np^i \\
&\le \sum_{i=0}^{\lceil\log_{1/p} n\rceil}(2+np^i) + \sum_{i=0}^{\infty}3p^i \\
&\le n/(1-p) + O(\log_{1/p} n) \\
&= O(n)
\end{aligned}$$

for any constant $0 < p < 1$.

For all three operations (search, insert, delete) the cost of operating on the key k is closely related to the length of the search path for k in a skip list of size n. Therefore, we analyze the expected length of the search path for an arbitrary key k. To analyze the cost of a search, we split the cost into two parts: The number of times the search follows a down pointer and the number of times the search follows a right pointer.

The first part, i.e., the number of times the search follows a down pointer is exactly h, the height of the list. Let

$$h_i = \begin{cases} 1 & \text{if } X^{(i)} \neq \emptyset \\ 0 & \text{if } X^{(i)} = \emptyset \end{cases}$$

and observe that $h_i \le X^{(i)}$. Then the expected height of the skiplist is given by

$$\begin{aligned}
\mathbf{E}\left[\sum_{i=0}^{\infty}h_i\right] &= \sum_{i=0}^{\infty}\mathbf{E}[h_i] \\
&= \sum_{i=0}^{\lceil\log_{1/p} n\rceil}\mathbf{E}[h_i] + \sum_{i=\lceil\log_{1/p} n\rceil+1}^{\infty}\mathbf{E}[h_i] \\
&\le \sum_{i=0}^{\lceil\log_{1/p} n\rceil}1 + \sum_{i=\lceil\log_{1/p} n\rceil+1}^{\infty}\mathbf{E}[|X^{(i)}|] \\
&\le 2+\log_{1/p} n + \sum_{i=\lceil\log_{1/p} n\rceil+1}^{\infty}\mathbf{E}[np^i] \\
&\le 2+\log_{1/p} n + \sum_{i=0}^{\infty}p^i \\
&\le \log_{1/p} n + O(1)
\end{aligned}$$

All that remains is to analyze the cost of following right pointers. Let r_i be the number of right pointers the search follows at level i. Again, we can always fall back on the trivial bound $r_i < X^{(i)}$. To get a more refined bound, imagine running the search backwards, starting from the node containing k in L_0 and working our way left and upwards to the top left node. When at a node v in L_i, if v appears in L_{i+1} then we go up, otherwise, we go to v's left neighbour. This process traces exactly the same search path, except in reverse.

For a node v that appears in L_i, the probability that it also appears in L_{i+1} is p, so the probability that this reverse path traverses exactly j edges at level i is at most $p(1-p)^j$. Therefore, the expected number of edges traversed at level i is at most

$$r_i \le \sum_{j=0}^{\infty} jp(1-p)^j = p\sum_{j=0}^{\infty} j(1-p)^j = (1-p)/p\ .^1$$

Therefore, the expected total number of (down and right) pointers we follow at all levels is at most

$$\mathbf{E}\left[h + \sum_{i=0}^{\lceil \log_{1/p} n \rceil} r_i + \sum_{i=\lceil \log_{1/p} n\rceil + 1}^{\infty} |X^{(i)}|\right] = (1+1/p)\log_{1/p} n + O(1)\ ,$$

for any constant p.

The expected cost of searching for the key k is proportional the expected length of the search path for k, which is $O(\log n)$. The cost of deleting the key k is proportional to the length of the search path for k, which is also $O(\log n)$. The cost of inserting a key k is proportional to the length of the search path for k in the skiplist we obtain after the insertion, which is $O(\log(n+1)) = O(\log n)$.

Theorem 9. *Skip lists support insertion, deletion, and searching of keys in* $O(\log n)$ *expected time and use* $O(n)$ *expected storage.*

Another nice property of skip lists is that the expected number of pointer updates during an insertion or deletion is $p/(1-p)$. Therefore, if we combine skip lists with the persistence paradigm of Chapter 3 we get a data structure for next element search queries that whose expected size is $O(n)$ and that answers next element search queries in $O(\log n)$ time.

4.4 Discussion and References

Fractional cascading has found many applications over the years, especially in computational geometry. Willard [7] and Luecker [5] both independently discovered the idea that iterated search could be solved in $O(k + \log n)$ time and used this idea to speed up searches in segment trees (Section 4.2). Edelsbrunner *et al* [4] applied fractional cascading to speed up a next-element search data structure based on decomposing the elements into monotone chains. Chazelle and Guibas' two part article [1, 2] describes a very general form of fractional cascading and its applications. Cole [3] uses fractional cascading to obtain his celebrated $O(\log n)$-time, $O(n)$-processor parallel merge-sort algorithm. Mehlhorn and Näher [6] show that fractional cascading can be done in a dynamic setting. That is, insertions and deletions to the individual lists can be done in $O(\log n \log\log n)$ time per operation and iterated search queries can be done in $O(\log n + h \log\log n)$ time.

^{1}Here we use the identity $\sum_{j=0}^{\infty} jx^j = x/(1-x)^2$, for any $x < 1$.

Bibliography

[1] B. Chazelle and L. J. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1(2):133–162, 1986.

[2] B. Chazelle and L. J. Guibas. Fractional cascading: II. Applications. *Algorithmica*, 1(2):163–191, 1986.

[3] R. Cole. Parallel merge sort. *SIAM Journal on Computing*, 17(4):770–785, 1988.

[4] H. Edelsbrunner, L. J. Guibas, and J. Stolfi. Optimal point location in a monotone subdivision. *SIAM Journal on Computing*, 15(2):317–340, 1986.

[5] G. S. Lueker. A data structure for orthogonal range queries. In *19th Annual Symposium on Foundations of Computer Science*, pages 28–34, 1978.

[6] K. Mehlhorn and S. Näher. Dynamic fractional cascading. *Algorithmica*, 5:215–241, 1990.

[7] D. E. Willard. *Predicate-Oriented Database Search Algorithms*. PhD thesis, Harvard University, 1978.