C.S. 252
Computational Geometry

Prof. Roberto Tamassia
Sem. II, 1992–1993

Fractional Cascading

Lecture 11
Scribe: Darren Erik Vengroff
Date: March 15, 1993

1 Introduction

Fractional cascading is a data structuring technique designed to support iterative searching in a set of k catalogs. It was first developed by Chazelle and Guibas [. Chazelle Guibas Data, Chazelle Guibas Applications .]. Fractional cascading is able to beat this method by using information obtained in a search of one catalog to to guide searches of subsequent catalogs.

2 The Iterative Search Problem

We should begin by formally defining the iterative search problem. We are given a set of k catalogs $C_1, C_2, \ldots, C_k$ over an ordered universe U of keys. We can think of a catalog as a finite ordered subset of U. The number of elements in catalog i is $|C_i| = n_i$. The total number of elements in all the catalogs is $n = \sum_i n_i$. We are allowed to preprocess the catalogs, and then asked to answer queries. A query is of the following form: given a query value x return the largest value less than or equal to x in each of the k catalogs. This is illustrated in Figure 1.

3 Three Approaches to Iterative Searching

There are at least three ways that we might consider approaching the problem of iterative searching in k catalogs. We could construct a search tree for each of the catalogs individually, we could try to construct a single search tree over the union of all the catalogs, or we could use fractional cascading. As we shall see, each of the first two techniques give us either optimal space or optimal space, but not both, whereas fractional cascading gives us both.

A lower bound on space usage for a data structure supporting iterative search is clearly $\Omega(n)$ since we must store the values of the n keys in the catalogs. A lower bound on query time is $O(\log n + k)$. The $\log n$ term follows from the decision tree model, while the k term follows from the need to report k separate results. As we shall see, fractional cascading will achieve both of these bounds.

The most naive way of performing an iterative search would be to construct a balanced search tree for each of the catalogs and then answer a query by searching each of the trees individually. Let us consider the complexity of this approach. As far as space is concerned,

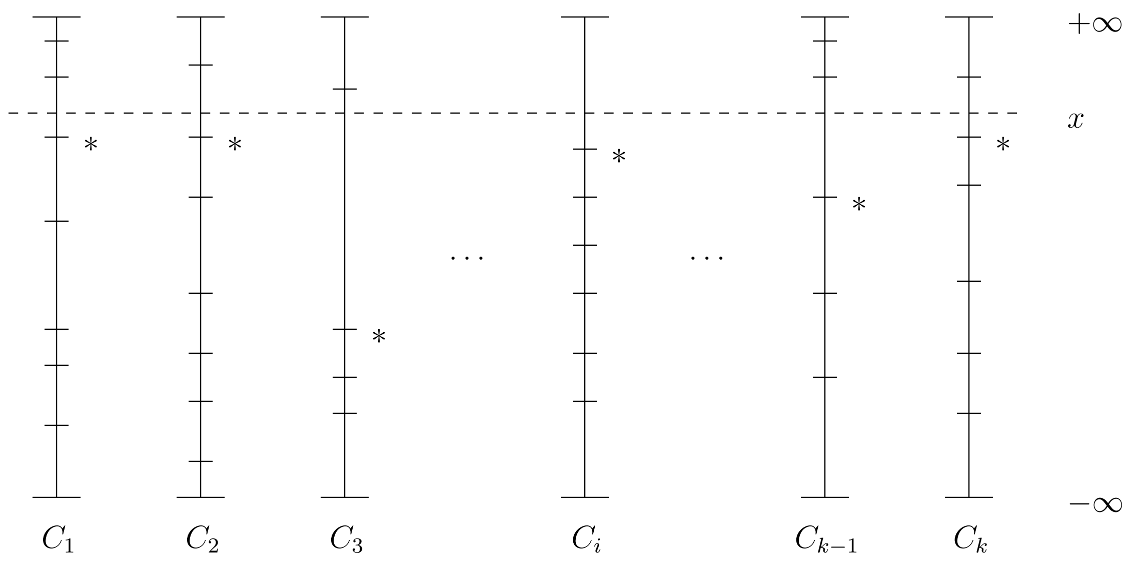

Figure 1: A simple iterative search query. The tick marks represent the values stored in the catalogs, including dummy values of $\pm\infty$. The query value x is shown as a horizontal dotted line. Each value returned by the query is marked with a $*$. Note that in each catalog the largest value less than or equal to x is returned.

the balanced tree on the ith catalog will occupy $O(n_i)$ space, thus the total space usage of all k such trees will be $\Theta(n)$. For query time, this approach will require a separate search in each of the k search trees. The size of each tree is bounded by $O(n)$, so the total search time is bounded by $O(k \log n)$. We can easily construct a worst case example where this bound is tight. Consider the case where $k = \sqrt{n}$ and each catalog contains $\sqrt{n}$ elements. The the total query time will be

$$k * \log \sqrt{n} = k * \frac{\log n}{2} = \Theta(k \log n).$$

The second approach is to construct a balanced search tree of the elements of the set $C = \bigcup_i C_i$. At each leaf of the tree, we store a k-tuple listing the largest element from each original catalog that is less than or equal to the value stored at the leaf. The space usage of this tree is $O(n)$ for the tree and $O(kn)$ for the k-tuples stored at the leaves. To perform a query in this data structure, we simply search the tree for the largest valued leaf less than or equal to x and then read of the answer stored there. The time to do this is $O(\log n)$ for the tree search and $O(k)$ for the reporting of the k-tuple, for a total of $O(\log n + k)$.

The third approach is fractional cascading, which achieves optimal query time using optimal space. The details of how this is done will comprise the remainder of this paper. The space and time usage of the three methods just described is summarized in Table 1.

4 Bridges

The central idea of fractional cascading is that of bridges. Intuitively, a bridge is a pointer from an element c_i of C_i to an element c_{i+1} of C_{i+1} where $|c_i - c_{i+1}|$ is small. This is illustrated

Method	Space	Time
k Search Trees	$O(n)$	$O(k \log n)$
Combined Search Tree	$O(kn)$	$O(\log n + k)$
Fractional Cascading	$O(n)$	$O(\log n + k)$

Table 1: Comparison of the time and space usages of the three iterative search methods described in Section 3.

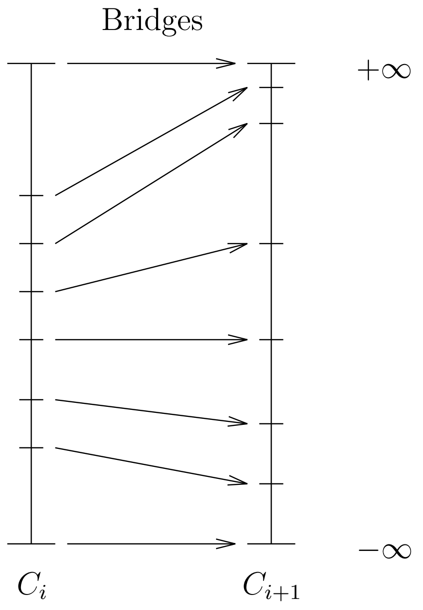

Figure 2: A series of bridges between two consecutive catalogs C_i and C_{i+1}.

in Figure 2. The idea is that once we have located the answer to a query in one catalog we should be able to follow a bridge to a key that is close to the answer in the next catalog. Ideally, we will be able to follow a bridge from the answer to the query in C_i to a key in C_{i+1} and then from there locate the answer in C_{i+1}, all in constant time. If we are able to do this, then once we have the answer in C_1 we can find the answer in the remaining $k-1$ catalogs in $O(k)$ time. To find the answer in C_1, we can just use a balanced binary search tree, thus making the overall time complexity of our algorithm optimal $O(\log n + k)$. By cleverly choosing where and how to construct bridges, this is exactly what fractional cascading will do.

Let us pause for a moment and consider a naive bridge building strategy and why it will not work. Suppose that we simply build a bridge from each element of each catalog to the element of the next catalog that is closest in value to it. Clearly this will not take any more than optimal space, since it will add only $O(n)$ bridges to the catalogs, whose total space is $O(n)$. Now, let us consider the time complexity of answering a query by looking at the worst case example illustrated in Figure 3. Here the query time is at least $\Omega(k \log n)$ even if the additional step of building balanced search trees over ranges of keys that fall between incoming bridges is taken.

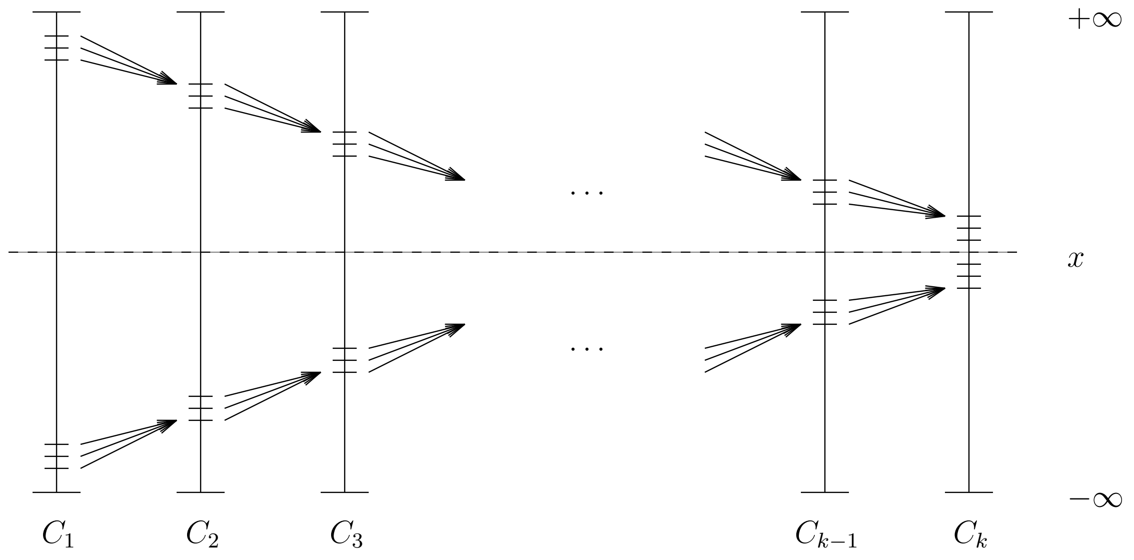

Figure 3: A worst case for the naive bridge building strategy. The query value is x. Even if each time we find an answer in catalog C_i we follow the bridge pointer from it as well as from the key above it we are still left with the entire contents of catalog C_{i+1} to search. This continues through the entire set of k catalogs. If we search each catalog by doing a linear scan from the point at which the bridge told us to begin, the total search time will be $\Omega(nk)$. Even if we build a balanced search tree over all elements in C_{i+1} that appear between two consecutive bridge pointers from C_i the query time will still be $\Omega(k \log n)$.

5 Bridges and Pointers for Fractional Cascading

As is now clear, we must be clever about how we choose bridges in order for fractional cascading to perform queries in optimal time. The way will do this is by augmenting each catalog C_i to produce an augmented catalog A_i. The augmentation is performed by by introducing synthetic keys. Synthetic keys are keys that were not originally part of a catalog C_i. Keys that were originally part of the catalog will be called real keys.

We build the augmented catalogs by beginning with A_k and proceeding backwards down to A_1. A_k is simple; it is exactly C_k. Now, for A_i where $1 \quad i < k$, we take every other key of A_{i+1} and add it to C_i as a synthetic key, then build a bridge from each such synthetic key to the element of A_{i+1} on which it was based. Additionally, we add bridges between the pseudo-elements $\pm\infty$ in consecutive catalogs. This process is best understood through an example, such as that appearing in Figure 4.

In addition to the bridges, each key in an augmented catalog A_i will have two pointers. If the key is a real key, then the first pointer points to the smallest synthetic key above it in A_i and the second pointer points to the largest key below it in A_i. It the key is a synthetic key, the pointers are similar, except that they point to the smallest real key above it and largest real key below it. If there is no key of the appropriate type above or below the key, the pointer points to the pseudo-key at $\pm\infty$, whichever is appropriate. Bridges and pointers for a set of catalogs are illustrated in Figure 5. Note that in the figure only half the pointers (those pointing downward) are shown. This is done for clarity; the interested reader should

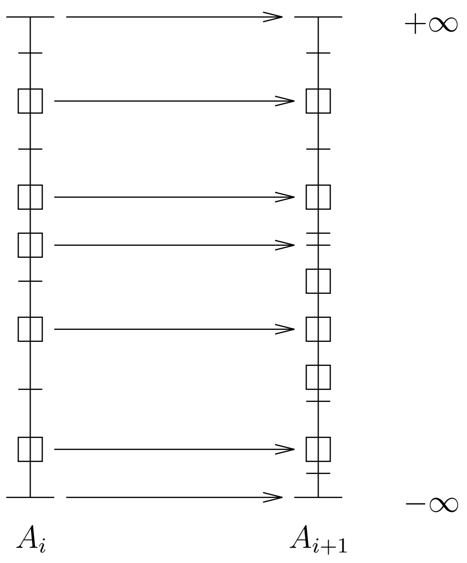

Figure 4: The construction of A_i. The real keys of A_i (elements of C_i) are shown as tick marks, while the synthetic keys (elements of $A_i \setminus C_i$) are shown as small squares. Note that the synthetic keys are simply copies of every other element of A_{i+1}.

be able to easily construct a similar set of upward pointing pointers.

An example of a *C++* implementation of the data structure representing a key in an augmented catalog appears in Figure 6. Compare this representation of a key to the pictorial representation in Figure 5 to get a better idea of exactly what is going on. This structure will be used in the code accompanying the discussion of the optimal iterative search algorithm in Section 6.

6 Fractional Cascading Search Algorithm

Given the data structure described in Section 5 it is relatively easy to construct an algorithm for iterative search that runs in optimal time. An fragment of am implementation of the algorithm is shown in Figure 7; it is based on the data structure from Figure 6.

7 Bridge and Pointer Space Complexity

8 Graph Structured Collections of Catalogs

References

.[]

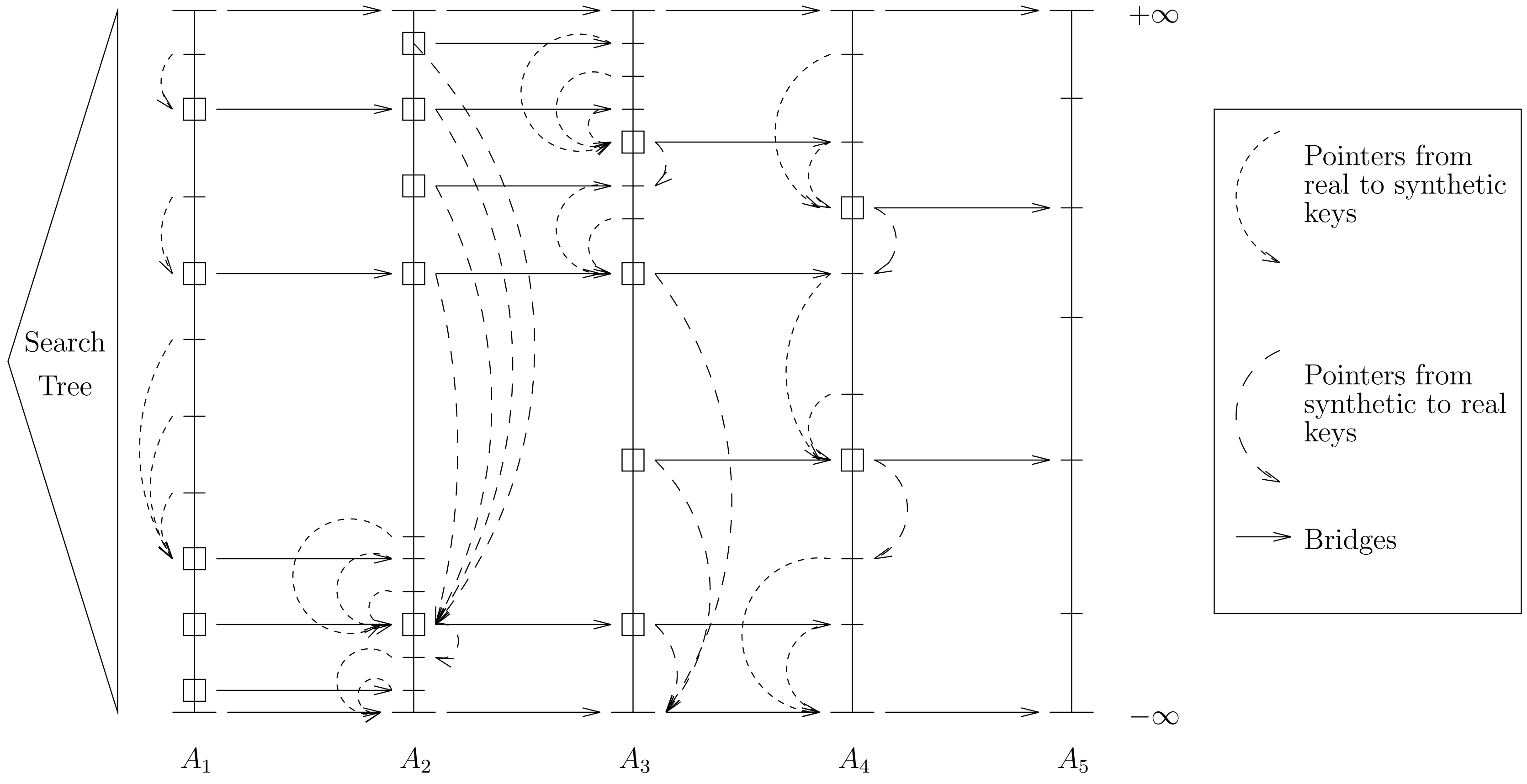
Search
Tree
$+\infty$
$-\infty$
A_1
A_2
A_3
A_4
A_5
Pointers from real to synthetic keys
Pointers from synthetic to real keys
Bridges

Figure 5: An example of a set of catalogs augmented for fractional cascading. As in Figure 4, tick marks represent elements of the original catalogs and squares represent synthetic keys added by bridging. The pointers that are shown illustrate how each synthetic key points to the largest valued key below it and similarly each real key points to the largest valued synthetic key below it. In addition to the pointers shown, each key has a pointer to the smallest valued key of the other type that is above it. These pointers are omitted for clarity. A balanced search tree whose leaves are the keys in A_1 is used to locate the key in A_i from which bridge traversal begins.

```
// Types of keys (real, synthetic, or special +-infinity key):

enum {
    KEY_TYPE_REAL,
    KEY_TYPE_SYNTHETIC,
    KEY_TYPE_PLUS_INFTY,
    KEY_TYPE_MINUS_INFTY
};


// The key structure:

struct key {
    int         type;           // KEY_TYPE_*

    keyvalue    keyval;         // The value stored at this key.

    key         *prev;          // The key immediately before this one
                                // in the catalog.  It can be of any type.
                                // If this->type is KEY_TYPE_MINUS_INFTY
                                // then this->prev == this.

    key         *jump_next;     // The key immediately after this one
                                // in the catalog if all keys having the
                                // same type as this are skipped.  The only
                                // exception to this rule is that if
                                // If this->type is KEY_TYPE_PLUS_INFTY
                                // then this->jump_next == this.

    key         *bridge;        // A bridge to the next catalog.  If
                                // type == KEY_TYPE_REAL then this is
                                // NULL.
};
```

Figure 6: A fragment of a *C++* implementation of the data structure used to implement a key in an augmented catalog. This structure will be used in the code accompanying the discussion of the optimal iterative search algorithm in Section 6.

```
// Search the collection of catalogs for a given key value.

int fc_catalog_collection::search(keyvalue x)
{
    int         ii;
    key         *current_key;

    // Locate the key to be reported from the first catalog.

    current_key = search_balanced_tree(x);

    // For every catalog report the key found and then locate a key in
    // the next catalog.

    for (ii = k-1; ii--; ) {

        my_assert((current_key->type == KEY_TYPE_REAL) ||
                  (current_key->type == KEY_TYPE_MINUS_INFTY),
                  "Bad key type to report.");

        current_key->report_me();

        // Follow the bridge below the current key to the next catalog.

        current_key = current_key->jump_below->bridge;

        // Now check the two keys above the current key to see if either
        // of them falls below the query value.  If so, take the highest
        // one.

        if (current_key->prev->keyval > x) {
            if (current_key->prev->prev->keyval > x) {
                current_key = current_key->prev->prev;
            } else {
                current_key = current_key->prev;
            }
        }

        // If the key we located is synthetic, then just jump to the
        // first real or -infinity key below it.

        if (current_key->type == KEY_TYPE_SYNTHETIC) {
            current_key = current_key->jump_next;
        }
    }

    // Now we just have to report the key from the final catalog
    // and we are done.

    my_assert((current_key->type == KEY_TYPE_REAL) ||
              (current_key->type == KEY_TYPE_MINUS_INFTY),
              "Bad key type to report.");

    current_key->report_me();

    return 0;
}
```

Figure 7: A fragment of a *C++* implementation of iterative search using fractional cascading.