Key indexed searching and sorting

Table of contents

Introduction

If the keys used in searching are integers only, we can use key-indexed searching to search in constant time! Compare that to the best one we've seen so far — $\log_2(n)$ in the case of binary search.

Let's say we have an array A of non-negative (why?) integers that we want to search for very quickly. If we think of implementing a set or a map (also called a dictionary), we can do the following: Use the values in A (the keys) as indices into another array B, which must have a capacity of one greater than the largest key in A. (Why?)

Let's say we're given an array `A = [5 1 7 2 8 4]`, and we want to quickly search for some `key` in `A`. We first create another array `B` which has a capacity of one larger than largest element in `A`, which is `8 + 1 = 9`. Then we populate `B` with the elements in `A` as indices into `B`. This is what we call the **setup** phase below, shown below.

```
      0   1   2   3   4   5
    -------------------------
A = | 5 | 1 | 7 | 2 | 8 | 4 |
    -------------------------

Create B, initially with all 0's in it.

      0   1   2   3   4   5   6   7   8
    -----------------------------------
B = | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
    -----------------------------------

Populate B, by using elements of A as indices into B.

      0   1   2   3   4   5   6   7   8
    -----------------------------------
B = | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
    -----------------------------------
```

```
For each element of A, the corresponding slot in B is set to 1.
```

Searching for any `key` can be done by simply indexing into `B` using the `key` as an index — if the slot is used (that is, the value is `1`), then the `key` exists in `A`; otherwise, it does not. Searching for `8` in `A`, we see that `B[8] == 1`, so `8` exists in `A`; searching for `6` in `A`, we see that `B[6] == 0`, so `6` does not exist in `A`; and so on.

Extending this to support non-negative integers is trivial, which we discuss at the end in Extensions.

We can also use this technique to sort in linear time! Compare that to the best we've seen so far — n^2 for bubble/selection/insertion sorts. An added restriction on the input is that the keys must be distinct (why?) in addition to being non-negative.

Let's say we have to sort the a `A = [5 1 7 2 8 4]` that we used in our search example. The **setup** phase is exactly the same — we create `B` and then populate it using the elements of `A` as indices into `B`. Now we can walk through `B` and output the elements that are set to `1`, and we get the sorted output.

```
Populated B, by using elements of A as indices into B.

      0   1   2   3   4   5   6   7   8
     -----------------------------------
B =  | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
     -----------------------------------

Iterate through B, and for each element that is 1, append the value of
the index to A.

      0   1   2   3   4   5
     -------------------------
A =  | 1 | 2 | 4 | 5 | 7 | 8 |
     -------------------------
```

Extending it to support non-negative integers is the same as in the case of search. Extending this to support non-distinct integers is possible, but only under certain conditions, which we discuss at the end in Extensions.

Back to Table of contents

Searching

INPUT: array of `n` non-negative distinct integers `A[0 ... n-1]`
INPUT: an non-negative integer `key` to search for
OUTPUT: True if the key exists, or false otherwise. The actual search must be done in constant time, even if building the intermediate data structure takes longer (linear for example).

Setup:

1. Find the maximum value `k` in the input, and create a temporary array B that has a capacity of `k + 1` (why the +1?), initializing the elements to 0 (automatically done in Java).

```
int k = A[0];
for (int i = 1; i < A.length; i++)
     if (A[i] > k)
           k = A[i];

int[] B = new int[k + 1];
```

2. Iterate through the keys in `A`, and using each element in `A` (key) as the index into `B`, set it to `1` (or any non-zero value).

```
for (int i = 0; i < A.length; i++)
    B[A[i]] = 1;
```

Search:

3. Once the array `B` is ready, search for a key takes constant time. If the key is outside the valid range, it's trivially false; otherwise, if the element indexed by the key is `1` it's true.

```
if (key < 0 && key >= B.length)
    return false;
else
    return B[key] != 0;
```

Search performance

For an input of `n` elements, building `B` takes `n` assignment operations, so the running time is a **linear function** of `n`. The actual search operation, after `B` is populated, is done in **constant time**! First of all, note that the algorithm is **out-of-place**, which means we have to consider the space overhead as well as running time. The extra space required is the capacity of `B`, which is equal to the maximum value in the input (`k` in step 1). If `k` is roughly of the order as `n`, the overhead is justified; however, if `k` is much larger than `n`, then the space overhead may not be justified. The space overhead is completely unjustified if your input is `[5, 13, 2, 10⁸, 9]` — for just `5` elements, you'll need a temporary array `B` with a capacity of `~10⁸`

!

Back to Table of contents

Sorting

We can also sort in linear time! Compare that to the n^2 sorting algorithms we've seen so far in this course. We'll see much better sorting algorithms later on, but still nothing as good as linear time.

INPUT: array of `n` non-negative distinct integers `A[0 ... n-1]`
OUTPUT: elements of A are in sorted in non-decreasing order.

Setup: Use steps 1 and 2 from the searching case to build the array B.

1. Same as step 1 in the search case above.
2. Same as step 2 in the search case above.

Sort:

3. Once the array B is ready, iterate through B and fill in A.

```
int i = 0;
for (int k = 0; k < B.length; j++) {
   if (B[k] != 0) {
       a[i++] = k;
   }
}
```

Back to Table of contents

Sort performance

Building `B` takes at most `n` operations (assignments in this case). Filling in `A` using `B` takes at most `k` assignments, where `k` is the maximum value in the input. So the running time is proportional to `n + k`, which means the running time depends on not just the size of the input (`n`), but also the magnitude of the input (`k`) — the first of its kind we've seen in this course. If `k` is roughly of the same order as `n`, then we have a **linear** sorting algorithm. However, if `k` is much larger than `n`, it may not be worth using this algorithm.

For the space overhead, the same argument as in the search case applies here as well.

Back to Table of contents

Extensions

How do we handle the case of negative keys? And, what about non-distinct keys? We discuss these two extensions below.

Back to Table of contents

Negative keys

We can easily accommodate negative keys by "shifting" the input such that the elements form valid array indices; the easiest way is to "shift" the input by the absolute of the minimum input value such that all the values are now positive (and thus valid array indices).

When searching for a key, we must first shift the key before using it as an index to see if it exists or not.

When sorting, we must shift the keys back when filling in `A` using `B`.

Back to Table of contents

Non-distinct keys

This is not a problem for searching, since we only need to know if a key is in the input or not, not how many times it occurs in the input. However, for sorting, the output will contain only one copy of each key, which means that the output will not be correct if the input contains non-distinct keys.

Let's change the setup (`B`) and sorting slightly to accommodate non distinct keys:

Setup:

1. Step 1 is unchanged.
2. Instead of just setting the elements in `B` to `0` (does not exist) or `1` (exists), we count the number of occurrences of the key (`>= 0`).

```
for (int i = 0; i < A.length; i++)
    B[A[i]] = B[A[i]] + 1;
```

Sorting:

3. Once the array `B` is ready, iterate through `B` and fill in `A`, keeping track of the number of copies of each key.

```
int i = 0;
for (int k = 0; k < B.length; j++) {
    while (B[k] > 0) {
        a[i++] = k;
        B[k] = B[k] - 1;
    }
}
```

Now we can sort an array of non-distinct integer keys. Will the output be **stable**?

While this works for input consisting of just integer keys, it does not work when we have key/value pairs. The multiple values corresponding to the same key will get overwritten by one of the values, so you get the wrong answer. Consider sorting the following key/value pairs: `[{9,"a"}, {5,"x"}, {2,"n"}, {5,"y"}, {9,"t"}]`. The array B as used here will not suffice, since it can just hold the key, not the value. If we change `B` to hold key/value pair, it will still hold a single value. We can of course extend `B` to be an array of lists, where the list keeps the multiple values for each key. The values must be sorted as well — either maintained as a sorted list, or sorted when filling `A`. Of course, the data structure just got a whole lot more complicated, and beginning to feel like a lot of work! This is the same **Adjacency List** data structure we used for representing a **graph** (coming in a few weeks) by the way. We will also use it for **hashtables**, also coming in a few weeks.

Back to Table of contents