Assignment 16, Data-Structures

Oleg Sivokon

<2016-05-25 Wed>

Contents

1 Problems

1.1 Problem 1

1. Given a hash-table with chaining of initial capacity m. What is the probability four elements inserted will end up in the same bucket?
2. Given a hash-table with open addressing and elements k_1, k_2 and k_3 inserted in that order, what is the chance of performing three checks when inserting the third element?
3. Given hash-table s.t. its density is $1-\frac{1}{\lg n}$. Provided the table uses open addressing, what is the expected time of failed search as a function of n?

1.1.1 Answer 1

Our simplifying assumption is that we draw hashing functions at random from a universe of hashing functions allows us to say that a probability of a key being hashed to a slot in the table of m slots is $\frac{1}{m}$. Using product law we can conclude that the probability of four keys being mapped to the same slot is $\frac{1}{m} \times \frac{1}{m} \times \frac{1}{m} = \frac{1}{m^3}$.

1.1.2 Answer 2

Using the same simplifying assumption as before, we see that for the element k_3 to be places only after three checks it has to first collide with k_1 and then with k_2. Using product law gives the probability of $\frac{1}{m} \times \frac{1}{m} = \frac{1}{m^2}$.

1.1.3 Answer 3

Recall that the average time needed for failed search in a hash table with open addressing is $\frac{1}{1-\alpha}$. Substituting $1 - \frac{1}{\lg n}$ in place of α obtains:

$$\begin{aligned}\frac{1}{1-\alpha} &= \frac{1}{1-\frac{1}{\lg n}} \\ &= \frac{\lg n}{\lg n - 1}\end{aligned}$$

1.2 Problem 2

Given a set of rational numbers S and a rational number z,

1. write an algorithm that finds two distinct summands of z with running time $\Theta(n)$.
2. Same as in (2), but for four summands and time $\Theta(n^2)$.

1.2.1 Answer 4

See the auxilary function `two_summands_of` in the following answer for illustration.

The basic idea is to put all numbers from S into a hash-table. Then loop over k_i the keys of the resulting hash-table and look for a key k_j, s.t. $k_j = z - k_i$. This lookup can be done in constant time while loop over the hash-table will take linear time. Thus the eintire alrogrithm will complete in time linear in number size of S.

1.2.2 Answer 5

The idea is to compute all possible sums of two distinct elements in S. This can be accomplished in quadratic time. Then we run modified

`two_summands_of` on the resulting sums. We modify it to allow for the same sum produced in different ways. It is again easy to see that the auxilary function `two_summands_of` runs only once and in linear time in the size of its input, that is quadratic in size of S, and the preparation step runs in quadratic time as well. Thus overall complexity of `four_summands_of` is quadratic in size of S.

```
list two_summands_of(int sum, chashtable summands) {
    iterator* it = (iterator*)make_hashtable_iterator(summands);
    do {
        pair kv = (pair)((printable*)it)->val;
        printable* a = kv->first;
        int s = *(int*)a->val;
        printable* b = (printable*)make_printable_int(sum - s);
        printable* found = chashtable_get(summands, b);
        if (found != NULL) {
            if (s * 2 == sum && ((list)kv->last)->cdr == NULL)
                continue;
            else if (s * 2 == sum)
                return cons(((list)kv->last)->car,
                            cons(((list)kv->last)->cdr->car, NULL));
            else return cons(kv->last, cons(found, NULL));
        }
    } while (next(it));
    return NULL;
}
```

```
list four_summands_of(int sum, array summands) {
    chashtable table = make_empty_int_chashtable();
    size_t i, j;
    for (i = 0; i < summands->length; i++) {
        printable* a = summands->elements[i];
        for (j = i + 1; j < summands->length; j++) {
            printable* b = summands->elements[j];
            printable* halfsum = printable_sum(a, b);
            pair elts = make_pair();
            elts->first = a;
            elts->last = b;
            list wrapper = cons((printable*)elts, NULL);
            printable* found = chashtable_get(table, halfsum);
            if (found != NULL)
                wrapper = cons((printable*)found, wrapper);
            chashtable_put(table, halfsum, (printable*)wrapper);
        }
    }
    list four = two_summands_of(sum, table);
    if (four != NULL) four = append(four->car, four->cdr->car);
    return four;
}
```

(Look for utility functions in `../lib`)

1.3 Problem 3

Given a binary search tree with n nodes there are $n + 1$ *left* and *right* nil-pointers. After performing the following on this tree: If `left[z]` = `nil`, then `left[z]` = `tree-predecessor(z)`, and if `right[z]` = `nil`, then `right[z]` = `tree-cussessor(z)`. The tree built in this way is called "frying pan" (WTF?), and the arcs are called "threads".

1. How can one distinguish between actual arcs and "threads"?
2. Write procedures for inserting and removing elements from this tree.
3. What is the benefit of using "threads"?

1.3.1 Answer 6

Search tree invariant implies that left pointer must point at a node with a value less than the node holding the pointer, but predecessor would have a value larger than the node holding the pointer. The situation for right node is symmetrical.

1.3.2 Answer 7

It's the same procedure you would use with a regular binary search tree, however instead of checking for `NULL` you would check if the value pointed at is less or greater, depending on side you are inserting the child at.

1.3.3 Answer 8

No difference what so ever. It doesn't matter whether the pointer points as some other node or nowhere. It makes it more difficult to write code to work with such trees, but that's about it.

1.4 Problem 4

Given array $A[1 \dots n]$ s.t.

$$\begin{aligned} A[1] > \cdots > A[p] \\ A[p+1] > \cdots > A[q] \\ A[q+1] > \cdots > A[n] \\ A[1] < A[q] \\ A[p+1] < A[n] \end{aligned}$$

insert it into binary tree.

1. What is the height of the resulting tree?
2. Erase $A[p+1]$ and insert it anew: how will the height and the shape of the tree change?

1.4.1 Answer 9

The height of the resulting tree is $\max(p, 1+q-p, 2+n-q)$. This is so because we will only insert right nodes at $A[p+1]$ and $A[q+1]$. After that all nodes inserted at the left side of the last right node inserted.

1.4.2 Answer 10

After reinserting $A[p+1]$ it will become right child of its former left child. This operation is essentially equivalent to left rotation, so it will increase the height of the tree by one if $p < 1+q-p \wedge q-p > n-q$, it will decrease it if $p < 1+q-p \wedge q-p < n-q$ and it will leave it unchanged if $p > 1+q-p \wedge p > 2+n-q$.