Hash tables for zip codes in C

Sam Serbouti serbouti@kth.se

October 19, 2024

In this assignment, I will build a hash table structure. They function very much like a dictionary : we look for an element using the key that is tied to it. This gives us, theoretically constant access time. However, in practice, if we want to have good memory management, it can get a bit trickier (especially concerning collisions). The goal here, is to implement a hash table for geographical `areas` : the keys are zip codes and they are linked to the area's name `char *name` and population number `int pop`. The codes are an `int n` tied to the `area *areas`.

1 Searching for a specific zip code

I start by building an array that holds the structure `area` and writing two versions of a search procedure.

Linear search

Go linearly (element by element) up to `postnr->n` and check:

```
if(strcmp(postnr->areas[i].zip,target_zip)==0){return &postnr->areas[i];}
```

Binary search

Start at `int left = 0` and `int right = postnr->n - 1` run a while loop and return `NULL` if I don't find it.

```
while(left<=right){
    int mid =left+(right-left)/2;
    int cmp =strcmp(postnr->areas[mid].zip,target_zip);
    if (cmp == 0){return &postnr->areas[mid];}
    if (cmp < 0){left= mid+1;}  //loo in right half
    else {right= mid-1;}  //look in left half
}
```

Converting char to int

Zip codes are numbers, we can see them as integers rather than strings (which were quite ugly to use) using `atoi` for conversion. This will save up memory space and computation time (it is faster to compare integers than strings). The `area` data structure becomes an `int zip`, a `char *name` and an `int pop`.

zip code	linear	binary
111 15	28	120
984 99	20473	135

Table 1: Runtime to find the zip codes (as integers) in ns

I observe that searching for the last item using binary is much faster (because we skip a lot of elements and linear $\hookrightarrow \mathcal{O}(log(n))$) than linear but linear is faster for the first zip code.

2 Using zip codes as array indices

Now, I want a constant access time. To achieve that, I will use the zip code as an index inside an array. Since the biggest coded is 99999, I will built an array with 100000 cells. The `area` doesn't change, but `codes` now holds an array of pointers of size `AREA`[1] : `area *areas[AREAS]` and an integer `n`.

```
area *lookup(codes *postnr, int zip) {
    if (postnr->areas[zip]!=NULL) {
        return postnr->areas[zip];
    }
    return NULL;//not found
}
```

zip codes as strings	direct	binary
111 15	237	1439999
984 99	73	656693

Table 2: Runtime to find the zip codes in ns

Binary search is normally used on dense, sorted arrays. But `postnr->areas` has a lot of zip codes that don't exist, that's why direct access is better.

A hash function

To take care of this bad memory management, I can try to add a function that could convert the zip code into an index. But sometimes, two keys can lead to the same index which is why I need to first test my dataset to see how many keys I should use to find a good compromise between the size of the keys array and the number of collisions.

[1]a constant set to 100000

To do this, I use a procedure that gives the collision frequency based on the hash table size (ie the modulo).

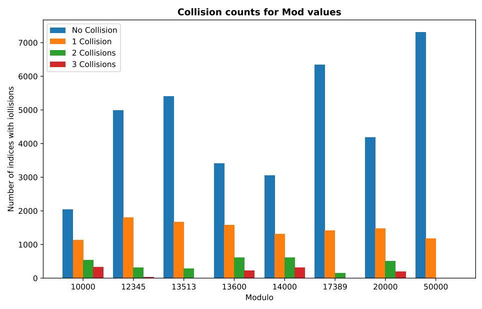


Figure 1: Mod = 17389 seems like the better tradeoff after this test

I notice that, overall, for smaller values of `mod`, there are more collisions, which is logical because there are fewer slots to put the keys in. Now, `key=zip%17389` (each zip code is given an index into a smaller array than the original one). And when I go through my dataset to build the data structure, I must use the hash to insert the area at the correct index:

```
int index=hash_function(a->zip);
postnr->areas[index]=*a;
postnr->n++;
```

3 Using buckets to handle collisions

When a collision happens, I want to be able to go though these elements and manually check which one is the correct one.

For this, I will use an array bucket : each possible result from the hash function (there are 17389) is linked to an array that is empty at first, but that is dynamically changed to add new elements when our zip codes match that hash value. Now, in the hash table, we no longer have the area but a pointer that leads to the bucket and when a collision happens I can resize the bucket's arrayto make room for more elements.

```
typedef struct area {
    int zip;
    char *name;
    int pop;
} area;
```

```
typedef struct bucket {
    area *entries;
    int nbOfElements;
    int nbOfSlots;
} bucket;
```

```
typedef struct codes {
    bucket *buckets;
    int n;//number of zips
} codes;
```

The new search and insert procedures

```c
void insert_to_bucket(codes ..
            ..*postnr, area *a){
    int index =hash_function(a->zip);
    bucket *b =&postnr->buckets[index];

    if (b->entries==NULL){
        b->entries=(area*)malloc(..
                ..sizeof(area));
        b->nbOfElements=0;
        b->nbOfSlots=1;
    }
    else if(b->nbOfElements>=b->nbOfSlots){
        b->nbOfSlots*=2;
        b->entries=(area*)realloc(..
     ..b->entries,b->nbOfSlots*sizeof(area));
    }
    b->entries[b->nbOfElements] =*a;
    b->nbOfElements++;
}
```

```c
area* lookup_in_bucket(codes *postnr, int zip) {
    int index = hash_function(zip);
    bucket *b = &postnr->buckets[index];
    if (b->entries == NULL) {return NULL;}
    for (int i = 0; i < b->size; i++) {
        if (b->entries[i].zip == zip) {
            return &b->entries[i]; //found :)
        }
    }
    return NULL;  //not found :(
}
```

When going through the dataset, I add the zips into the buckets with `insert_to_bucket(postnr, a)` after creating the area structures[2]

Functionality test

Trying to see if my program can insert and search for zip codes that do and don't exist:

```
search 11115 (exists): found zip: 11115 name: STOCKHOLM pop:3
search 59500 (does not exist):  not found
```

Searchtime benchmark

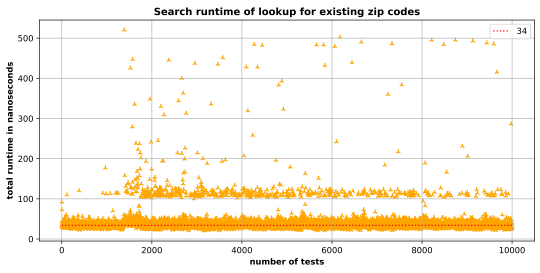


Figure 2: 10000 random searchtimes $\approx 34ns \hookrightarrow \mathcal{O}(1)$ it's perfect :)

[2]meaning, after allocating memory space, getting the zip code integer, the name and population size

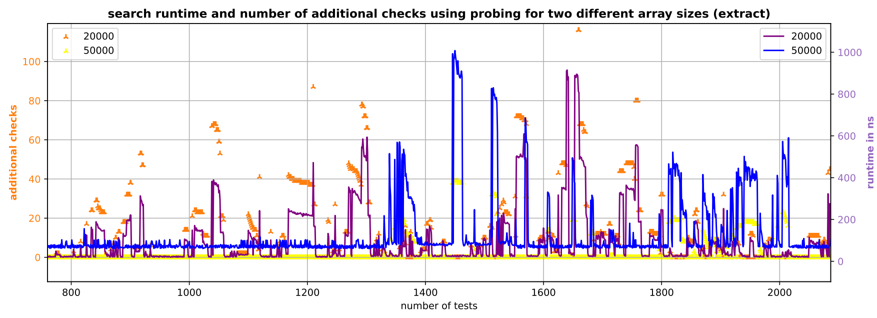
4 How 'bout no buckets?

So far, I am using plenty of buckets... I could just search for the next empty array slot and add the element there (hoping it won't be far away). This is called linear probing, or open addressing.

Of course, if the array is too small to begin with, this method isn't the one to choose. I picked my array to be 20000 big. But the idea is fairly simple: starting at `index = hash_function(a->zip, postnr->size)`, if there is a collision, I start incrementing the `index=(index+1)%postnr->size` until `postnr->areas[index].zip != -1`[3]. If I want to insert an area:`postnr->areas[index]=*a` and if I want to lookfor an area: I check `if (postnr->areas[index].zip == zip)`.

Tests

The functionality test is the same as before and gave the same results: it found 11115 after 0 additional checks and it did not find 59500 after 2 additional checks.



The benchmark shows how bigger arrays lead to less additional checking for neighbouring empty slots in the array (which makes sense: the bigger the array, the less collisions happen) and whenever there is a need for searching for empty slots, the runtime increases.

Conclusion

The idea behind hash tables isn't that far from using regular arrays. However the devil is in the details of optimizing memory space and computation time. Overall, the strength of this data structure is all about finding the best array size for the hash table. But we could also change the hash function, use linked lists for the buckets.. I would say it depends on what type of data set we are using and how/how often we expect them to change.

[3]ie we found an empty slot