Hash tables

in C

Algorithms and data structures ID1021

Johan Montelius

Spring 2026

Introduction

This assignment will show you the most natural way to organize a set of entries that should be accessible given a key. We will start by using a not so efficient solution, then move to something that is fast but does waste space and hopefully land in something that is both space and time efficient.

A table of zip codes

To explore different techniques we will use a table of Swedish zip codes. The original file is in a CSV format (comma separated value). We will read the file and insert each item in a array of entries. Each entry consists of the zip code, the name of the area and the population. The following skeleton code will get you started reading the file and adding them all to an array (there are 9675 entries in total).

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

typedef struct codes {
  area *areas;
  int n;
} codes;
```

Reading a file and creating the codes data structure is quite complicated but this procedure will do the trick. The constant `AREAS` is set to 10000 i.e.

enough to hold the areas described in the file. The constant BUFFER is set to 200; that should be enough to read a line.

The procedure `getline()` reads a line from the file an places it in the buffer. If the line is longer than the buffer it will allocate a larger buffer and update the value `n`. Since the area needs its own strings we copy the buffer to a copy. The copy is then divided into three parts using `strtok()` (the delimiter that we are looking for is replaced by `'0'` thereby dividing the string into three strings.

```
codes *read_postcodes(char *file) {

  codes *postnr = (codes*)malloc(sizeof(codes));

  area *areas = (area*)malloc(sizeof(area)*AREAS);

  FILE *fptr = fopen(file, "r");

  int k = 0;

  char *lineptr = malloc(sizeof(char)*BUFFER);
  size_t n = BUFFER;

  while((k < AREAS) && (getline(&lineptr, &n, fptr) > 0)) {
    char *copy = (char*)malloc(sizeof(char)*n);
    strcpy(copy, lineptr);

    area a;
    a.zip = strtok(copy, ",");
    a.name = strtok(NULL, ",");
    a.pop = atoi(strtok(NULL, ","));

    areas[k++] = a;
  }
  fclose(fptr);

  postnr->areas = areas;
  postnr->n = k;

  return postnr;
}
```

Note that the areas array is an array of data structures i.e. not an array of pointers to data structures. In C we have an option to do either and we need to make a decision.

Now write a lookup method that does a linear search through all zip codes looking for a specific entry. Then, since the zip codes in the file are ordered, you can write a binary search method that does the same. Write a small benchmark that searches for "111 15" and "984 99" and explain the results.

Since we know that all zip codes are numbers we might as well convert them to Integers before creating the entries. Create a new version of your zip program where you change the area data structure to hold an integer as code and then populate the data as follows:

```
char *zip = strtok(copy, ",");
a.zip = atoi(zip)*100 + atoi(zip+3);
```

Re-run all benchmarks and presents the results, has the execution time improved?

Use key as index

Let's do something different. If we have the zip code as the key and the key is an integer, why not use that integer as an index in an array? We know that the highest possible key is 99999 so why not construct an array that is a hundred thousand elements large and then use the key as index, perfect let's go.

The only thing you need to change is to increase the size of the data array, change how we populate the array and then implement a `lookup()` method. Run the benchmarks and again and compare the time to do a lookup to the binary search method.

size matters

The only drawback with the implementation that you have now is that the array is to 90% empty. You have an array of a hundred thousand elements but there are less then ten thousand zip codes. This might not be a big problem since we are only talking about some hundred thousand bytes but in general this could of course be a huge drawback.

The solution is to somehow transform the original key into an index in a smaller array. If we can find a function that takes a zip code key and returns an index in the range 0 to let's say 10000 then the problem would be solved. The function could not be too time consuming since the whole point is to save time so it should be very simple.

The function that transforms a key to an index is called a hash function. One simple way of defining a hash function is to simply take the key modulo some value m in hope that the indexes should be fairly unique. If we have two keys that maps to the same index then we have *collision* that is something

that we need to handle (and will be able to handle) but the fewer collisions the better.

Do an experiment where you read all the zip codes from the file and then run through them creating an index modulo m for some values of m (10000, 20000 ...). Your experiment should count the number of collisions of each type i.e. two keys map to the same index, three keys map to the same index etc. The following skeleton code should get you starting:

```
void collisions(codes *postnr, int mod) {

  int mx = 20;

  int data[mod];
  int cols[mx];

  for(int i = 0; i < mod; i++) {
    data[i] = 0;
  }

  for(int i = 0; i < mx; i++) {
    cols[i] = 0;
  }

  for (int i = 0; i < postnr->n; i++) {
    int index = (postnr->areas[i].zip)%mod;
    data[index]++;
  }

  int sum = 0;
  for(int i = 0; i < mod; i++) {
    sum += data[i];
    if (data[i] < mx)
      cols[data[i]]++;
  }

  printf("%d (%d) : ", mod, sum);
  for (int i = 1; i < mx; i++) {
    printf("%6d ",cols[i]);
  }
  printf("\n");
}
```

Do some runs with growing number of modulo operator. Apart from testing values like 10000 and 20000 try something like 12345 or 17389, any difference?

Finding a hash function is always a trade off between the size of the array (the maximum index) and the number of collisions. The larger array that is used the less collisions will you likely have but the more space is wasted. A larger array does not necessarily mean that we will have less collisions. Try the following three values: 13513, 13600 and 14000 - did the larger size help?

handling collisions

When you have a hash function without too many collisions, it's time to learn how to handle these. One simple solution is to have an array of *buckets*, each bucket holds a small set of elements that all have the same hash value. The bucket can be implemented as a linked list of codes or a small array of codes. The linked list might be easier to implement since you do not have to care about adding too many codes to the bucket. In an array you do need to handle the problem of allocating a slightly larger array when a collision occurs.

All elements that are added with out a collision only induce an extra reference (the indirection from the array to the bucket) and minimal memory overhead.

Implement a version of you program that uses an array of buckets. The array is of course initially empty and only when an entry is added do you allocate the minimal bucket. The lookup procedure must of course check that the element actually has the correct zip code or find the element with the correct zip code if we have several in the bucket. Even if there is only one element in the bucket can you trust the hash function that this is the element that you're looking for. You might do a lookup of zip code that does not exist yet the hash function gives you an index that is the also the index of a legal zip code.

slightly better?

A slightly more efficient (but take care) version of the bucket implementation is to use the array itself without indirection to separate buckets. The trick is to start with the hashed index and then move forward in the array to find the right entry. The lookup procedure would stop as soon as it finds a empty slot and the hope is that this should not take too long. This is true if the array is sufficiently large for example twice as large as needed. If the array is too tight the risk is of course that hundred of elements needs to be examined (and what will happen if the array is full?).

Implement the improved version and do some statistics on how many element you need to look at before finding the one that you're looking for. Try with an increasing size of the array and compare the results with the original solution.