Hash tables in C

Ying Pei Lin

Fall 2024

Zip Code Table

At `read_postcodes` function, we store the zip codes in the format of char and after we use the `strcmp` function to compare the zip codes in the search function. Down below is the code snippet of the search function.

```c
area* linear_search_char(codes *postnr, const char *zip) {
  for (int i = 0; i < postnr->n; i++) {
    if (strcmp(postnr->areas[i].zip_char, zip) == 0) {
      return &postnr->areas[i];
    }
  }
  return NULL;
}

area* binary_search_char(codes *postnr, const char *zip) {
  int left = 0;
  int right = postnr->n - 1;

  while (left <= right) {
    int mid = (left + right) / 2;
    int comparison_result = strcmp(postnr->areas[mid].zip_char, zip);

    if (comparison_result == 0) {
      return &postnr->areas[mid];
    }
    if (comparison_result < 0) {
      left = mid + 1;
    } else {
      right = mid - 1;
    }
  }
  return NULL;
}
```

This is not efficient because the `strcmp` function compares the zip codes character by character. We can improve the search function by converting the zip codes to integers and compare them directly. The Table 1 shows the time taken for linear search with different data types while Table 2 shows the time taken for binary search with different data types.

ZIP Code	Search Type	Time (ns)
”111 15”	Linear Search	3 ns
”111 15”	Binary Search	39 ns
”984 99”	Linear Search	25037 ns
”984 99”	Binary Search	36 ns

Table 1: Comparison the Search Times for Char Data Type ZIP Codes in Linear and Binary Searches

ZIP Code	Search Type	Time (ns)
111 15	Linear Search	0 ns
111 15	Binary Search	18 ns
984 99	Linear Search	5076 ns
984 99	Binary Search	32 ns

Table 2: Comparison the Search Times for Integer Data Type ZIP Codes in Linear and Binary Searches

Before comparing the results, we should note that these two cases are the edge cases. The first case 111 15 is the first element in the array, and the second case 984 99 is the last element in the array.

For the first case, the linear search is faster than the binary search because the linear search can find the element in the first iteration. For the second case, the binary search is faster than the linear search because it does not have to iterate through all the elements to find the last element. Additionally, the data type does affect the search time. The integer data type is faster than the character data type in both linear and binary searches.

Direct Indexing

The direct indexing method is a way to improve the search time by using the zip code as an index to the array. To achieve this, I add a new `areas_direct` array is an array of pointers to the `area` struct and use the zip code as the index to the array. The modified `codes` struct is as follows:

```
typedef struct codes {
  area *areas;
  area **areas_direct; // Array of pointers to areas
  int n;
} codes;
```

To initialize the direct indexing, I create a new function `init_direct` to convert the zip code to an integer and use it as the index to the array after store the post codes using the original `read_postcodes` function.

```
codes *init_direct(codes *postnr) {
  postnr->areas_direct = (area**)malloc(sizeof(area*)*100000);
  for(int i = 0; i < postnr->n; i++) {
    int zip = zip_to_int(postnr->areas[i].zip_char);
    postnr->areas_direct[zip] = &postnr->areas[i];
  }
  return postnr;
}
```

Comparing the search time for the direct indexing method with the binary search methods, the direct indexing method is faster than the binary search method as shown in Table 3. The direct indexing method is faster because it uses the zip code as an index to the array, which is a constant time operation.

ZIP Code	**Search Type**	**Time (ns)**
111 15	Direct lookup	0 ns
111 15	Binary Search	18 ns
984 99	Direct lookup	0 ns
984 99	Binary Search	20 ns

Table 3: Comparison the Search Times for Integer Data Type ZIP Codes in Direct lookup and Binary Searches

Collisions

The direct indexing method is efficient when there are no collisions. However, when there are collisions, it will overwrite the previous element in the array. To handle collisions, we can use a linked list to store the elements with the same index. Here, I use a array of pointers to the `node` struct to store the elements with the same index. The method is called bucket hashing.

I modified the `codes` struct to include the `buckets` array of pointers to the `node` struct. The modified `codes` struct is as follows:

```
typedef struct bucket {
  area *areas;         // Dynamic array of areas
```

```c
  int size;             // Current size of bucket
  int capacity;         // Capacity of bucket
} bucket;

typedef struct codes {
  bucket **buckets;      // Array of pointers to buckets
  int n;                 // Number of areas
  int size;              // Size of hash table
} codes;
```

To insert the elements into the hash table, I use the `hash` function to convert the zip code to an integer and use it as the index to the array. No matter if the collision happens or not, the element will be added to the bucket. If the bucket is full, I resize the bucket by doubling the capacity.

```c
int hash(int zip, int size) {
  return zip % size;
}

void insert_bucket(codes *postnr, area *a) {
  int index = hash(a->zip_int, postnr->size);

  // If bucket doesn't exist, create it
  if(postnr->buckets[index] == NULL) {
    postnr->buckets[index] = malloc(sizeof(bucket));
    postnr->buckets[index]->capacity = 1;
    postnr->buckets[index]->size = 0;
    postnr->buckets[index]->areas = malloc(sizeof(area));
  }

  // If bucket is full, resize it
  bucket *bucket = postnr->buckets[index];
  if(bucket->size >= bucket->capacity) {
    bucket->capacity *= 2;
    bucket->areas = realloc(bucket->areas, bucket->capacity * sizeof(area));
  }

  // Add the area
  b->areas[b->size] = *a;
  b->size++;
}
```

The search function is similar to the direct indexing method, but it has to iterate through the bucket to find the element.

```
area* hash_lookup(codes *postnr, int zip) {
  int index = hash(zip, postnr->size);
  bucket *bucket = postnr->buckets[index];

  for(int i = 0; i < bucket->size; i++) {
    if(bucket->areas[i].zip_int == zip) {
      return &bucket->areas[i];
    }
  }
  return NULL;
}
```

The efficiency of the bucket hashing method depends on the size of the hash table and the number of collisions. If the hash table is too small, there will be many collisions, and the search time will increase. If the hash table is too large, there will be many empty buckets, and the memory usage will increase. Comparing the amount of collisions between different size of hash table, the Table 4 shows the number of collisions for different size of hash table(measured using the modified `collisions` function).

Size of Hash Table	**Numbers of Collision**					
	0	**1**	**2**	**3**	**4**	**5**
13513	5410	1678	287	12	0	0
13600	3406	1578	613	229	56	13
14000	3055	1316	615	320	134	31

Table 4: Comparison of the Number of Collisions for Different Sizes of Hash Table

Linear Probing

Another method to handle collisions is linear probing. In linear probing, if there is a collision, the element is added to the next available slot in the array. The modified `codes` struct is as follows:

```
typedef struct linear_hash {
  area *areas;      // Array twice the size needed
  int size;         // Size of hash table
  int count;        // Number of elements stored
} linear_hash;

typedef struct codes {
  area *areas;
  linear_hash *linear; // Linear probing hash table
  int n;                    // Number of areas
  int size;                 // Size of hash table
} codes;
```

To insert the elements into the hash table, I use the `hash` function to convert the zip code to an integer. If the slot is occupied, we move to the next slot until we find an empty slot to insert the element. The `init_linear_hash` function is as follows:

```
codes *init_linear_hash(codes *postnr, int size) {
  postnr->linear = malloc(sizeof(linear_hash));
  postnr->linear->areas = calloc(size * 2, sizeof(area));  // Double size for bette
  postnr->linear->size = size * 2;
  postnr->linear->count = 0;

  // Insert areas
  for(int i = 0; i < postnr->n; i++) {
    area a = postnr->areas[i];
    int index = hash(a.zip_int, postnr->linear->size);
    while(postnr->linear->areas[index].zip_int != 0) {
      index = (index + 1) % postnr->linear->size;
    }
    postnr->linear->areas[index] = a;
    postnr->linear->count++;
  }
  return postnr;
}
```

The search function is similar to the bucket hashing method, but it has to iterate through the array to find the element. First, we calculate the index using the hash function. Then we start from the index and move to the next slot until we find the element. To prevent infinite loops, we check if the number of probes is greater than the size of the hash table. If it is, we will return `NULL`.

```
area* lookup_linear(codes *postnr, int zip) {
  linear_hash *linear = postnr->linear;
  int probes = 0;
  int index = hash(zip, linear->size);

  while(linear->areas[index].zip_int != 0) {
    if(linear->areas[index].zip_int == zip) {
      return &linear->areas[index];
    }
    index = (index + 1) % linear->size;
    probes++;
    if(probes >= linear->size) return NULL;
  }

  return NULL;
}
```