6 Hash Tables

6.1 Objectives

The learning objectives of this laboratory session are:

- To understand the challenges of implementing *hash tables*.
- To get hands-on experience with implementing hash tables.
- To develop critical thinking concerning implementation decisions for hash tables.

The outcomes for this session are:

- Improved skills in C programming using pointers and dynamic memory allocation.
- A clear understanding of hash tables, the operations they support, and their use.
- The ability to make implementation decisions concerning hash tables.
- The ability to decide when hash tables are suitable in solving a problem.

6.2 Brief Theory Reminder

Table Types

A *table* is a collection of elements of the same kind, which are identified by *keys*. The elements stored in a table are also called *records*.
Tables may be organized in two ways:

- *Fixed* tables, where the number of elements to be stored is known at the moment of creation. A reserved keyword table for a programming language is an example of a fixed table. It is typically arranged as a pointer table, where pointers indicate reserved keywords in alphabetic order. Binary search is used when looking for a particular keyword.
- *Dynamic* tables, with a variable number of elements. Dynamic tables may be organized as: singly linked lists, search trees or hash tables. If dynamic tables are arranged as lists, then search is performed linearly, which slows down the process. A search tree reduces the time to find an element, but if it is not balanced, its worst case performance is also linear (i.e. $O(n)$).

A *hash table* (also called a *hash map*) is a data structure used to implement an *associative array* that can map keys to values. A hash table uses a *hash function* to compute an index into an array of *buckets* or *slots*, where the data records are stored and retrieved based on their keys.

Ideally, the hash function will assign each key to a unique bucket, but it is possible that two keys will generate an identical hash causing both keys to point to the same bucket. When such a situation occurs, we say that the two keys *collide*. Instead, most hash table designs assume that hash collisions will occur and must be accommodated in some way.

In a well-dimensioned hash table, the average cost (number of algorithm simple steps) for each lookup is independent of the number of elements stored in the table.

In many situations, hash tables turn out to be more efficient than search trees or any other table lookup structure. For this reason, they are widely used in many kinds of computer software, particularly for associative arrays, database indexing, caches, and sets. Hash tables are used if the keys are alphanumeric because comparisons are time-consuming for long alphanumeric keys (remember that hash tables construct shorter keys).

Hash Functions

A *hash function* is a function which transforms a key into a natural number called a *hash value*, i.e.

$$f : K \to H,$$

where K is the set of keys and H is a set of natural numbers. Function f is a many-to-one function. If two different keys, say k_1 and k_2, $k_1 \neq k_2$ have the same hash value, i.e. $f(k_1) = f(k_2)$, then the two keys are said to *collide* and the corresponding records are called *synonyms*. Two restrictions are imposed on f:

1. For any $k \in K$ the value should be obtained as fast as possible (i.e. no complicated computations).
2. It must minimize the number of collisions (i.e. distribute as evenly as possible).

An example of hash function is:

$$f(k) = \gamma(k) \text{ modulus } B,$$

where γ is a function which maps a key to a natural number, and B is a natural number, possibly prime. The expression of function γ depends on the keys. If the keys are numeric, the $\gamma(k) = k$. The simplest function γ on alphanumeric keys is the sum of the (ASCII) codes for each character of the key. A simple function on strings is shown in Listing 6.1:

Listing 6.1: A simple hash function.

```
#define B ? /* ? stands for a suitable value for B, the size of the bucket table */
int f(char *key)
{
  int sum, len;
  sum = 0;
  len = strlen(key);
  for (int i = 0; i < len; i++)
    sum += key[ i ];
  return(sum % B);
}
```

Collision Resolution

If collision occurs (multiple keys map to the same integer), then elements with different keys may be stored in the same "slot" of the hash table. Thus, when the hash function is used to locate a potential match, it will be necessary to compare the key of that element with the search key. But there may be more than one element which should be stored in a single slot of the table. Various techniques are used to manage this problem:

- *chaining* (chain all records with colliding keys in lists attached to the appropriate slot),
- overflow areas (linked list constructed in special area of table called overflow area),
- *re-hashing* (*open addressing*), using neighboring slots (linear probing), quadratic probing, random probing, etc.

With collision resolution by chaining (also called open hashing, as the records are stored in lists outside the table) we insert all records with colliding keys in a singly-linked list. There will be a number of lists, called *buckets* one for each hash value. Figure 6.1 illustrates the concept. In this figure, the records were inserted in the following possible order: records with keys i, j or k were the first, then records with keys i_1 or k_2, then record with key k_1. There is a linked list attached to each primary table slot. When inserting a record, we calculate the hash value of its key, and insert it in the list for that bucket. To look for a key, e.g. k_1, we calculate the hash value, $h(k_1)$ and follow the linked list. If we find the key in that list, then the record is in the table. If we reach the end of the list for that hash value, then the record is not in the table.

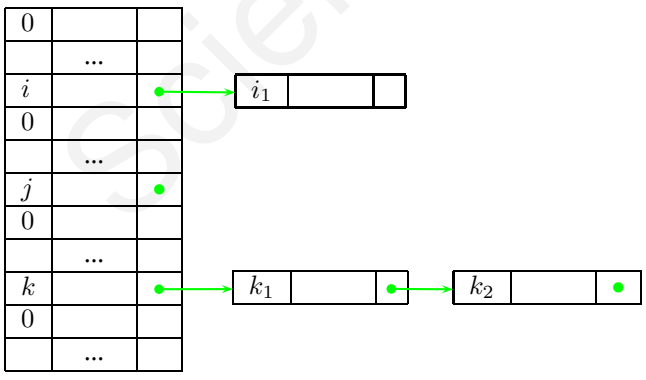

Figure 6.1: Collision resolution by chaining.

When using an overflow area for collision resolution, a linked list is constructed in a special area of table called overflow area (the actual chaining uses indexes in the bucket table). Figure 6.2 illustrates this strategy. When adding records with non-colliding keys, as it is with records with keys k_1 or k_3, when we apply the hash function, we find empty slots, and insert the records in those empty slots. But, when record with key k_2 comes to be inserted, its hash value $h(k_2)$ it is the same as $h(k_1)$: they collide. Then we get the first free slot in the overflow area, and insert the record with key k_2 in that slot. Searching is doen in the same way as in a linked list.

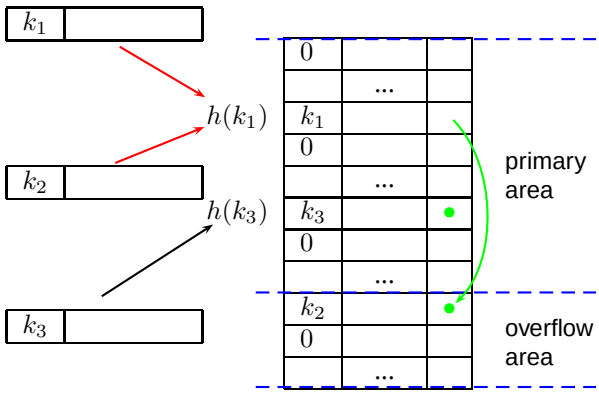

Figure 6.2: Collision resolution using overflow tables.

With *rehashing*, when a collision occurs, we try using a second hash function. There are many variations of this strategy, but the general term is re-hashing. As shown in Figure 6.3, when we add record with key k_1, we use the first hash function, h, and store the record with key k_1 in the available slot $h(k_1)$. When attempting to insert the record with key k_3, a collision occurs (with key k_1). Then, we apply a second hash function h', repeatedly, to find another place to insert. We find $h'(k_3)$ empty and insert the record with key k_3 there. When we search for a key, we use function h first, and then function h'.

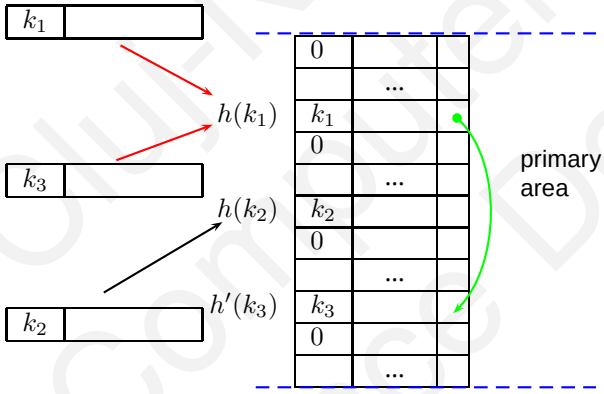

Figure 6.3: Collision resolution using rehashing.

In what follows, we will detail the implementation for collision resolution by chaining. For this organization, a cell in a bucket may have the type as shown in Listing 6.2.

Listing 6.2: A possible bucket record and bucket table declaration.

```
typedef struct cell
{
  char key[60];
  /* other useful info */
  struct cell *next;
} NodeT;

NodeT *BucketTable[B];
```

Initially, all the buckets are empty. This may be achieved with a simple loop, as shown in Listing 6.3.

Listing 6.3: Initializind the table to empty.

```
for (int i = 0; i < B; i++)
  BucketTable[i] = NULL;
```

The steps needed to find a record of hash value key in a hash table are shown below (Listing 6.4 shows a possible implementation, where the keys are strings):

1. Determine the hash value for the key $h = f(key)$.

2. Linearly search the bucket for hash value h:

Listing 6.4: Looking for a string key in a hash table.

```
// assumes that p is declared before, and that the hash value is stored in variable h
p = BucketTable[h];
while (p != NULL)
{
  if (strcmp(key, p->key) == 0)
    return p;
  p = p->next;
}
return NULL; /* not found */
```

A hash table is built by repeatedly inserting records into it.
Insertion in a hash table takes the following steps:

1. Make a new node pointed to by p and fill it:
2. Calculate the hash value:
3. If the bucket table header entry is empty, insert as first list element. A possible implementation of these steps is shown by Listing 6.5.

Listing 6.5: Steps in inserting a new record in a hash table.

```
p = (NodeT *) malloc(sizeof(NodeT));
if (p)
{
        fillNode(p);
        h = f(p->key);
        if (BucketTable[ h ] == NULL)
        { // empty slot
                BucketTable[ h ] = p;
                 p->next = NULL;
        }
}
else
// suitable message for no memory error
```

4. Otherwise check if the candidate record (record to insert) is not already present. If the candidate record is found in the table then an update may be applied. If the record is not present, then we may insert it as the first node in the bucket selected by the hash function (cf. Listing 4):

```
q = find(p->key);
if (q == 0)
{ /* not found. Insert it */
  p -> next = BucketTable[ h ];
  Buckettable[ h ] = p;
}
else /* double key */
  processRec(p, q);
```

A complete list of a hash table contents may be retrieved using code similar to the one shown in Listing 6.6.

Listing 6.6: Listing the contents of a hash table.

```
for (i = 0; i < B; i++)
  if (BucketTable[ i ] != NULL)
  {
    printf("Bucket for hash value %d\n", i);
    p =BucketTable[ i ];
    while (p != NULL)
    {
      showNode(p);
      p = p->next;
    }
  }
```

6.3 Laboratory Assignments

6.3.1 Mandatory Assignments

Write modular programs in C, using files for input and output, to:

6.3.1. Manage a hash table, using *collision resolution by chaining*, where the keys are student full names. .
I/O description. Input:

```
i<name>
d<name>
f<name>
l
```

`i<name>`=insert `<name>`, `d<name>`=delete `<name>`, `f<name>`=find `<name>` in the table; `l`=list table contents. Note that the characters `<`, and `>` are *not* part of the input. Use a hash function suitable for character strings and motivate your answer in a comment stored in the heading area of your hash table processing routines. Output for find should be `yes`, followed by the table index if found or `no` if not found.

6.3.2. Manage a hash table, using *open addressing*, where the keys are student full names. Provide create, insert, find, delete, and list operations on that table. .
I/O description. Input:

```
i<name>
d<name>
f<name>
l
```

`i<name>`=insert `<name>`, `d<name>`=delete `<name>`, `f<name>`=find `<name>` in the table; `l`=list table contents. Note that the characters `<`, and `>` are *not* part of the input. Use a hash function suitable for character strings and motivate your answer in a comment stored in the heading area of your hash table processing routines. Output for find should be `yes`, followed by the table index if found or `no` if not found.

6.3.2 Optional Assignments

6.3.3. Manage a hash table, using *open addressing*, where the keys are book ISBNs. Provide create, insert, find, delete, and list operations on that table. .
I/O description. Input:

```
i<name>
d<name>
f<name>
l
```

`i<name>`=insert `<name>`, `d<name>`=delete `<name>`, `f<name>`=find `<name>` in the table; `l`=list table contents. Note that the characters `<`, and `>` are *not* part of the input. Use a hash function suitable for character strings and motivate your answer in a comment stored in the heading area of your hash table processing routines. Output for find should be `yes`, followed by the table index if found or `no` if not found.

6.3.4. Manage a hash table, using *open addressing*, with character string keys, where the hash function should be selectable before each run. The methods you should use in building your has functions are linear, quadratic and double hashing (cf. Lecture 3). Your code should provide create, insert, find and delete operations on that table. .
I/O description. Input:

```
f<number>
i<name>
d<name>
f<name>
l
```

`f<number>`=select the hash function numbered with <number>, `i<name>`=insert `<name>`, `d<name>`=delete `<name>`, `f<name>`=find `<name>` in the table; `l`=list table contents. Note that the characters `<`, and `>` are *not* part of the input. Output for find should be `yes`, followed by the table index if found or `no` if not found.

6.3.5. Manage a hash table, using *open addressing*, with numeric long integer keys, where the hash function should be selectable before each run. The methods you should use in building your has functions are linear, quadratic and double hashing. Provide create, insert, find, delete, and list operations on that table. .
I/O description. Input:

```
f<number>
i<name>
d<name>
f<name>
l
```

`f<number>`=select the hash function numbered with <number>, `i<name>`=insert `<name>`, `d<name>`=delete `<name>`, `f<name>`=find `<name>` in the table; `l`=list table contents. Note that the characters `<`, and `>` are *not* part of the input. Output for find should be `yes`, followed by the table index if found or `no` if not found.

6.3.6. Manage a hash table, using *collision resolution by chaining*, with numeric long integer keys, where the hash function should be selectable before each run. The methods you should use in building your has functions are memory address, integer cast, and component sum. Provide create, insert, find, delete, and list operations on that table.. .
I/O description. Input:

```
f<number>
i<name>
d<name>
f<name>
l
```

`f<number>`=select the hash function numbered with <number>, `i<name>`=insert `<name>`, `d<name>`=delete `<name>`, `f<name>`=find `<name>` in the table; `l`=list table contents. Note that the characters `<`, and `>` are *not* part of the input. Output for find should be `yes`, followed by the table index if found or `no` if not found.

6.3.7. Manage a hash table, using *collision resolution by chaining*, with numeric long integer and string keys, where the hash function should use *polynomial accumulation*. Provide create, insert, find, delete, and list operations on that table. .
I/O description. Input:

```
i<name>
d<name>
f<name>
l
```

`i<name>`=insert `<name>`, `d<name>`=delete `<name>`, `f<name>`=find `<name>` in the table; `l`=list table contents. Note that the characters `<`, and `>` are *not* part of the input. Output for find should be `yes`, followed by the table index if found or `no` if not found.
