Chapter 9

TABLES AND INFORMATION RETRIEVAL

1. Introduction: Breaking the $\lg n$ Barrier
2. Rectangular Arrays
3. Tables of Various Shapes
4. Tables: A New Abstract Data Type
5. Application: Radix Sort
6. Hashing
7. Analysis of Hashing
8. Conclusions: Comparison of Methods
9. Application: The Life Game Revisited
Introduction: Breaking the $\lg n$ Barrier

- By use of key comparisons alone, it is impossible to complete a search of $n$ items in fewer than $\lg n$ comparisons, on average (lowerbound_search_thm).
- Ordinary table lookup or array access requires only constant time $O(1)$.
- Both table lookup and searching share the same essential purpose, that of information retrieval. The key used for searching and the index used for table lookup have the same essential purpose: one piece of information that is used to locate further information.
- Both table lookup and searching algorithms provide functions from a set of keys or indices to locations in a list or array.
- In this chapter we study ways to implement and access various kinds of tables in contiguous storage.
- Several steps may be needed to retrieve an entry from some kinds of tables, but the time required remains $O(1)$. It is bounded by a constant that does not depend on the size of the table. Thus table lookup can be more efficient than any searching method.
- We shall implement abstractly defined tables with arrays. In order to distinguish between the abstract concept and its implementation, we introduce:

```
Convention
The index defining an entry of an abstractly defined table
is enclosed in parentheses,
whereas the index of an entry of an array
is enclosed in square brackets.
```
In mathematics a **function** is defined in terms of two sets and a correspondence from elements of the first set to elements of the second. If \( f \) is a function from a set \( A \) to a set \( B \), then \( f \) assigns to each element of \( A \) a unique element of \( B \).

The set \( A \) is called the **domain** of \( f \), and the set \( B \) is called the **codomain** of \( f \).

The subset of \( B \) containing just those elements that occur as values of \( f \) is called the **range** of \( f \).
**Definition** A *table* with index set $I$ and base type $T$ is a function from $I$ to $T$ together with the following operations.

1. **Table access**: Evaluate the function at any index in $I$.
2. **Table assignment**: Modify the function by changing its value at a specified index in $I$ to the new value specified in the assignment.
3. **Creation**: Set up a new function.
4. **Clearing**: Remove all elements from the index set $I$, so there is no remaining domain.
5. **Insertion**: Adjoin a new element $x$ to the index set $I$ and define a corresponding value of the function at $x$.
6. **Deletion**: Delete an element $x$ from the index set $I$ and restrict the function to the resulting smaller domain.
Start with an array that holds the hash table.

Use a hash function to take a key and map it to some index in the array. This function will generally map several different keys to the same index.

If the desired record is in the location given by the index, then we are finished; otherwise we must use some method to resolve the collision that may have occurred between two records wanting to go to the same location.

To use hashing we must (a) find good hash functions and (b) determine how to resolve collisions.
Choosing a Hash Function

- A hash function should be easy and quick to compute.

- A hash function should achieve an even distribution of the keys that actually occur across the range of indices.

- The usual way to make a hash function is to take the key, chop it up, mix the pieces together in various ways, and thereby obtain an index that will be uniformly distributed over the range of indices.

- Note that there is nothing random about a hash function. If the function is evaluated more than once on the same key, then it must give the same result every time, so the key can be retrieved without fail.

- Truncation: Sometimes we ignore part of the key, and use the remaining part as the index.

- Folding: We may partition the key into several parts and combine the parts in a convenient way.

- Modular arithmetic: We may convert the key to an integer, divide by the size of the index range, and take the remainder as the result.

- A better spread of keys is often obtained by taking the size of the table (the index range) to be a prime number.
C++ Example of a Hash Function

We write a hash function in C++ for transforming a key consisting of eight alphanumeric characters into an integer in the range

\[ 0 \leq \text{hash} \leq \text{hash\_size} - 1. \]

We start with a class Key with the methods and functions of the following definition:

```cpp
class Key: public String {
public:
    char key_letter(int position) const;
    void make_blank();
    // Add constructors and other methods.
};
```

We inherit the methods of the class String from Chapter 6. The method `key_letter(int position)` must return the character in a particular position of the Key, or return a blank if the Key has length less than n. The final method `make_blank` sets up an empty Key.

```cpp
int hash(const Key &target) {
    /* Post: target has been hashed, returning a value between 0 and hash\_size – 1.
    Uses: Methods for the class Key. */

    int value = 0;
    for (int position = 0; position < 8; position++)
        value = 4 * value + target.key_letter(position);
    return value % hash_size;
}
```

We have simply added the integer codes corresponding to each of the eight characters, multiplying by 4 each time.
Linear Probing:

*Linear probing* starts with the hash address and searches sequentially for the target key or an empty position. The array should be considered circular, so that when the last location is reached, the search proceeds to the first location of the array.

Clustering:

Quadratic Probing:

If there is a collision at hash address $h$, *quadratic probing* goes to locations $h + 1$, $h + 4$, $h + 9$, $\ldots$, that is, at locations $h + i^2 \pmod{\text{hashsize}}$ for $i = 1, 2, \ldots$.

Other methods:

- Key-dependent increments;
- Random probing.
Hash Table Specifications

```cpp
const int hash_size = 997; // a prime number of appropriate size
class Hash_table {
    public:
        Hash_table();
        void clear();
        Error_code insert(const Record &new_entry);
        Error_code retrieve(const Key &target, Record &found) const;
    private:
        Record table[hash_size];
};
```

**Hash_table::Hash_table();**

*Post:* The hash table has been created and initialized to be empty.

**void Hash_table::clear();**

*Post:* The hash table has been cleared and is empty.

**Error_code Hash_table::retrieve(const Key &target, Record &found) const;**

*Post:* If an entry in the hash table has key equal to target, then found takes on the value of such an entry, and success is returned. Otherwise, not_present is returned.
Hash Table Insertion

Error_code Hash_table :: insert(const Record &new_entry)

/** Post: If the Hash_table is full, a code of overflow is returned. If the table already contains an item with the key of new_entry a code of duplicate_error is returned. Otherwise: The Record new_entry is inserted into the Hash_table and success is returned.**/

Uses: Methods for classes Key, and Record. The function hash. */
{
    Error_code result = success;
    int probe_count, // Counter to be sure that table is not full.
        increment, // Increment used for quadratic probing.
        probe; // Position currently probed in the hash table.
    Key null; // Null key for comparison purposes.
    null.make_blank();
    probe = hash(new_entry);
    probe_count = 0;
    increment = 1;

    while (table[probe] != null) // Is the location empty?
        && table[probe] != new_entry // Duplicate key?
        && probe_count < (hash.size + 1)/2) { // Has overflow occurred?
        probe_count++;
        probe = (probe + increment) % hash.size;
        increment += 2; // Prepare increment for next iteration.
    }
    if (table[probe] == null) table[probe] = new_entry;
        // Insert new entry.
    else if (table[probe] == new_entry) result = duplicate_error;
    else result = overflow; // The table is full.
    return result;
}
The linked lists from the hash table are called *chains*.

If the records are large, a chained hash table can save space.

Collision resolution with chaining is simple; clustering is no problem.

The hash table itself can be smaller than the number of records; overflow is no problem.

Deletion is quick and easy in a chained hash table.

If the records are very small and the table nearly full, chaining may take more space.
Code for Chained Hash Tables

Definition:

```cpp
class Hash_table {
public:
   // Specify methods here.
private:
   List<Record> table[hash_size];
};
```

The class List can be any one of the generic linked implementations of a list studied in Chapter 6.

Constructor:

The implementation of the constructor simply calls the constructor for each list in the array.

Clear:

To clear the table, we must clear the linked list in each of the table positions, using the List method clear().

Retrieval:

```cpp
sequential_search(table[hash(target)], target, position);
```

Insertion:

```cpp
table[hash(new_entry)].insert(0, new_entry);
```

Deletion:

```cpp
Error_code Hash_table :: remove(const Key_type &target, 
                                 Record &x);
```

Post: If the table has an entry with key equal to target, a code of success is returned, the entry is deleted from the hash table and recorded in x. Otherwise a code of not_present is returned.
1. Use top-down design for your data structures, just as you do for your algorithms.

2. Before considering detailed structures, decide what operations on the data will be required.

3. For the design and programming of lists, see Chapter 6.

4. Use the logical structure of the data to decide what kind of table to use.

5. Let the structure of the data help you decide whether an index function or an access table is better for accessing a table of data.

6. In using a hash table, let the nature of the data and the required operations help you decide between chaining and open addressing.

7. Hash functions must usually be custom-designed for the kind of keys used for accessing the hash table.

8. Recall from the analysis of hashing that some collisions will almost inevitably occur.

9. For open addressing, clustering is unlikely to be a problem until the hash table is more than half full.