The searching algorithms in Chapter 7 use only binary key comparisons as they traverse a dictionary in search of an item with a given key. Since the search keys are represented in computer memory, though, there is no need to restrict search algorithms to use only key comparisons. In this chapter we shall see some search algorithms that perform other arithmetic operations on keys besides binary comparisons. These address calculation techniques can be an excellent ways to maintain a static or dynamic dictionary.

8.1
PERFECT HASHING

Suppose an application needs to search by name for one of the planets in our solar system. We could store the names in alphabetical order in an array and use binary search to find any planet name with at most \( \lceil \log_2(9+1) \rceil = 4 \) string comparisons. Program 8.1 uses a very different strategy to store planet names so that any name can be found by computing a single function.

Program 8.1 stores planet names in a hash table. Macro \( h(s) \) in Program 8.1 computes a hash function that uses arithmetic operations to scatter key values through the slots in the hash table. In Figure 8.1, the debugging output from Program 8.1, we see that each key hashes to a different slot; we say that \( h(s) \) computes a perfect hash function on the names of the planets.
8.2 Collision Resolution Using a Probe Strategy

To resolve collisions, insert them into the dictionary and if a collision happens use chaining, which the hash function can do. For example, a hash table that uses probing with chaining, it would be impossible to delete a dictionary's hash. We would not know in advance which keys would be stored in the table. So we need to store a dynamic dictionary when we know about how to store a perfect hash function for another set of keys.

Program 8.1 probably seems unduly complicated. It gives no clue

Figure 8.1

where all elements of
sequence for key is hash(key)+1, hash(key)+2, hash(key)+3,...
The probe function

8.2 Collision Resolution Using a Probe Strategy

A program that uses perfect hashing on the names of the nine planets.

Program 8.1

```c

int main()

```
A hash table that is maintained with a probe strategy, say, Chaplin, is the

\[ D: \text{beep} \]
\[ B: \text{boop} \]
\[ C: \text{beep} \]
\[ B: \text{boop} \]
\[ A: \text{beep} \]
\[ C: \text{boop} \]

When we insert a new element, we follow the insertion algorithm until we reach an empty slot, then put the new element into that slot.

The sequence of events that the above sequence of events are taken modulo the size of the hash table. Algorithm 8.3 collision resolution using a probe strategy.

```
Algorithm 8.3

1. Insert, if found and table full, return with the following message:
   "You have exceeded the maximum number of elements in the table."
2. Insert, if not found, return with the following message:
   "The element was not found in the table.
   - Enter the element at the current position in the table.
   - If the table is full, return with the following message:
     "The table is full."
   - Else, return with the following message:
     "The element was inserted successfully.
     - The element is now at the current position in the table.
   - If the table is empty, return with the following message:
     "The table is empty."
```

1. Insert, if found and table full, return with the following message:
   "You have exceeded the maximum number of elements in the table."
2. Insert, if not found, return with the following message:
   "The element was not found in the table.
   - Enter the element at the current position in the table.
   - If the table is full, return with the following message:
     "The table is full."
   - Else, return with the following message:
     "The element was inserted successfully.
     - The element is now at the current position in the table.
   - If the table is empty, return with the following message:
     "The table is empty."

```
Algorithm 8.3 collision resolution using a probe strategy:

1. Insert, if found and table full, return with the following message:
   "You have exceeded the maximum number of elements in the table."
2. Insert, if not found, return with the following message:
   "The element was not found in the table.
   - Enter the element at the current position in the table.
   - If the table is full, return with the following message:
     "The table is full."
   - Else, return with the following message:
     "The element was inserted successfully.
     - The element is now at the current position in the table.
   - If the table is empty, return with the following message:
     "The table is empty."
```
In Figure 8.3, function (8) is plotted along with the data obtained from the experiment.

\[
\left(\frac{n-1}{n+1}\right)^\frac{1}{2}
\]

Figure 6.7 |

approximately
collision resolution using linked lists

8.3

The scheme the compressor in section 8.2 would appear in these lists.

Further than three, the lists themselves in the slots of a hash table, we can make the slots contain pointers to linked lists of items. Under this scheme, the compressor in section 8.2 would appear in these lists.

/* Load Factor a

<table>
<thead>
<tr>
<th>Load Factor a</th>
<th>0.2</th>
<th>0.4</th>
<th>0.6</th>
<th>0.8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>6</td>
<td>12</td>
<td>24</td>
</tr>
</tbody>
</table>

The arrows show the graph of function (6.1). The dark line shows the expected length of a successful search.

The expected length of an unsuccessful search in a hash table filled to a of its capacity is approximately 1/(1-a) for every slot in the table. The expected length of a successful search takes one probe. But an unsuccessful search must examine every slot in the table. The expected length of a successful search takes one probe. But an unsuccessful search must examine every slot in the table.

Collision resolution strategies that use probe sequences have a large table and long, expensive probes. If we use a hash table to store a key's dynamic dictionary, then we can store a key's dynamic dictionary. The expected length of an unsuccessful search in a hash table filled to a of its capacity is shown in Figure 8.2.

Statically or randomly determined hash functions were

\[
\text{Hash}(h(0), \ldots, h(n)) = \sum_{i=0}^{n} (h(i) + 1) \mod \text{Table Size}
\]

The expected length of an unsuccessful search path when we use

expressions (8.1) and (8.2) are both approximations. Indeed, they

\[
\left[ \frac{z}{1 + z} \right] = \frac{z}{1 + z} = \frac{1}{1 + z}
\]

General Problems with Probe Strategies

They exhibit terrible performance. They want to get close to 1, because when hash tables are nearly full, they can't be anything but unsuccessful searches, so that as is not too close to 1.

Successful and unsuccessful searches as long as is not too close to 1. Both expressions are good approximations to the expected lengths of unsuccessful searches when is large, when the table is full. However, they
the hash table, we need to examine other items when we seek a specific item. This is called a collision, and it occurs when two different items are hashed to the same slot in the table.

To handle collisions, two common techniques are chaining and open addressing. In chaining, we create a linked list of items that hash to the same slot. In open addressing, we look for the next available slot to insert the item when a collision occurs.

The choice between these techniques depends on the characteristics of the data being stored. Chaining is often used when the probability of collisions is low, while open addressing is preferred when the load factor (the ratio of the number of items to the table size) is high.

In this chapter, we will explore the concept of hashing and its applications. We will cover the principles of hashing, the design of hash functions, and the implementation of hash tables. We will also discuss the use of hashing in real-world applications, such as database indexing and file systems.
Good Hash Functions

The worst case of hashing is easily realized by the constant hash function. In this case, all keys would hash to the same value, and all keys would have the same address. This would make it impossible to search for a key in the hash table. Therefore, it is important to use a good hash function that distributes the keys evenly across the hash table.

A simple but effective hash function is the modulo function. For example, if we want to use a hash table of 1000 buckets, we can use the hash function h(k) = k mod 1000. This will distribute the keys evenly across the hash table, and it will ensure that no two keys hash to the same value.

However, this simple hash function can lead to collisions, where two or more keys hash to the same value. To resolve collisions, we can use techniques such as chaining or open addressing.

Chaining involves using a linked list to store the elements that hash to the same value. Open addressing, on the other hand, involves using an array to store the elements. When a collision occurs, the next available slot is used to store the element.

In the worst case, all keys hash to the same value, and all elements would be stored in the same slot. However, in the average case, the expected number of elements stored in each slot is much lower. This is why hash functions and collision resolution techniques are important.
function:

13. Comment on the suitability of each of the following as a hash

12. Use a hash table to solve the problem in Section 1.4

11. Algorithm B.2 is subjected to delete an item from a hash table that

10. In C's collected hashing, use interpolation to decide a good value for

9. In C's collected hashing, use linear probing to decide a good value for

Algorithm B.2

```
[ ] insert(key) = [ ]
break
((i > l) || (i > r) || (i > j) || (i > k)) = [ ]
restriction

(y = ) = [ ] return
result

(0 == ) = [ ] return
result

mod UW+I+i = [ ]

(1) while

i = [ ]

/ * Select the position * / 0 = [ ]

/* Insert item in position i */
```

REFERENCES

EXERCISES

191