

## Lecture 8: Search and Hash Tables

### Course Learning Outcomes (CLOs):

- Use advanced searching techniques, such as **hashing** and 2-3 trees.
- Show how time and space complexities can be traded-off (e.g. distribution counting sort, **hashing**).

**Dr. Khalil Yousef**

Adopted from the Slides of the ECE 608 Computational Models and Methods Course at Purdue University

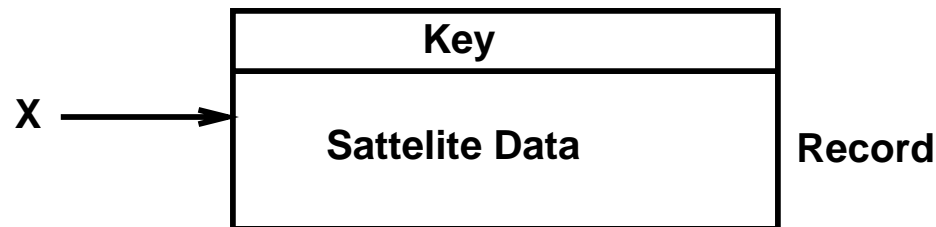
Review Chapter 10 and Read Chapter 11 of *Introduction to Algorithms*

## Sets

---

A fundamental concept in computer science is the dynamic **set**, which can grow and shrink over time. For example, a **dictionary** is a dynamic set that supports the insertion and deletion of elements, as well as a membership test.

Each element in a dynamic set is represented as an object whose fields can be examined or modified if we have a pointer to the object. Elements often have a **key field** as well as **satellite data**. For example:



The best way to implement a set depends on the operations that are supported on the set. It is also important to consider the number of elements that will be stored in the set  $S$ ,  $|S|$ , and compare it to the size of the universe  $U$ ,  $|U|$  (the set of elements that could conceivably be in  $S$ ).

## Operations on Sets

---

Operations on sets are either **queries** or **modifying** operations. The running time of these operations is reported in terms of the size of the set.

- **SEARCH( $S, k$ )**: a query that returns a pointer  $x$  to an element in  $S$  such that  $key[x] = k$  or NIL if there is no such element in  $S$ .
- **INSERT( $S, x$ )**: a modifying operation such that  $S \leftarrow S \cup \{x\}$ .
- **DELETE( $S, x$ )**: a modifying operation such that  $S \leftarrow S - \{x\}$ .
- **MINIMUM( $S$ )**: a query on a totally ordered set  $S$  such that a pointer to the element with the smallest key in  $S$  is returned.

## Operations on Sets *continued*

---

- **MAXIMUM( $S$ )**: a query on a totally ordered set  $S$  such that a pointer to the element with the largest key in  $S$  is returned.
- **SUCCESSOR( $S, x$ )**: a query on a totally ordered set  $S$  such that a pointer to the smallest element in  $S$  greater than  $x$  is returned, unless  $x$  is the maximum, in which case NIL is returned. This can easily be extended to sets with duplicate keys.
- **PREDECESSOR( $S, x$ )**: a query on a totally ordered set  $S$  such that a pointer to the largest element in  $S$  less than  $x$  is returned, unless  $x$  is the minimum, in which case NIL is returned. This can easily be extended to sets with duplicate keys.

## Set Examples

---

- The login table for a computer system with 100 users where logins are of length 10 and consist of lower case alphabetic characters;  $|U| = 26^{10}$ ,  $|S| = 10^2$ . Support INSERT, DELETE, and SEARCH.

## Implementation Issues for Dictionaries

---

One way to **implement a dictionary** is as a **linear list**. In this case, SEARCH can take  $\Theta(n)$  time in the worst case, and at least one of the INSERT or DELETE operations can take  $\Theta(n)$  time in the worst case.

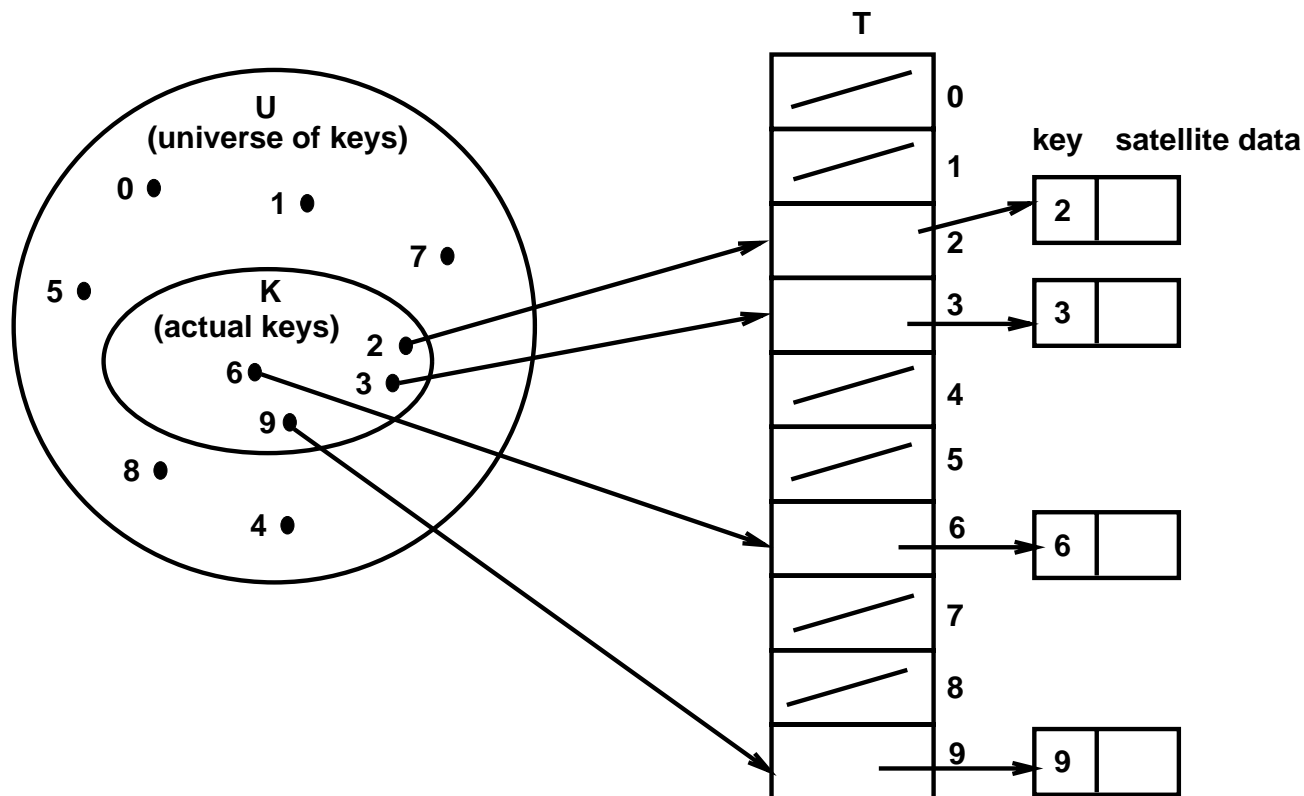
One way to **implement a dictionary** is as a **heap**. In this case, the INSERT and DELETE operations can take  $\Theta(\lg n)$  time in the worst case. What about the running time of SEARCH?

We **would like to use** a **datastructure that supports efficient dynamic set operations**. In particular, we would like the INSERT, SEARCH, DELETE operations to take  $O(1)$  time.

## The Direct-Address Table

To represent a dynamic set, we can use a direct-address table  $T[0..m-1]$  where each slot corresponds to a key in the universe and:

$$T[i] = \begin{cases} x & \text{if } x \in S \text{ and } \text{key}[x] = i, \\ \text{NIL} & \text{otherwise.} \end{cases}$$



## The Direct-Address Table *continued*

---

All of the following operations on a direct-address table are  $O(1)$ :

1. DIRECT-ADDRESS-SEARCH( $T, k$ )  
    **return**  $T[k]$

2. DIRECT-ADDRESS-INSERT( $T, x$ )  
     $T[key[x]] \leftarrow x$

3. DIRECT-ADDRESS-DELETE( $T, x$ )  
     $T[key[x]] \leftarrow \text{NIL}$

Direct Addressing is a simple technique that works well if the universe of keys,  $U = \{0, 1, 2, \dots, m-1\}$ , is relatively small and no two elements have the same key.



## Hash Tables

---

**Problem:** usually the range of keys is much larger than the desired table size  $m$  (e.g., ASCII strings).

A **hash table** is a generalization of an ordinary array which supports the direct addressing of an element in the table in  $O(1)$  average case time, although the worst case time is  $\Theta(n)$ .

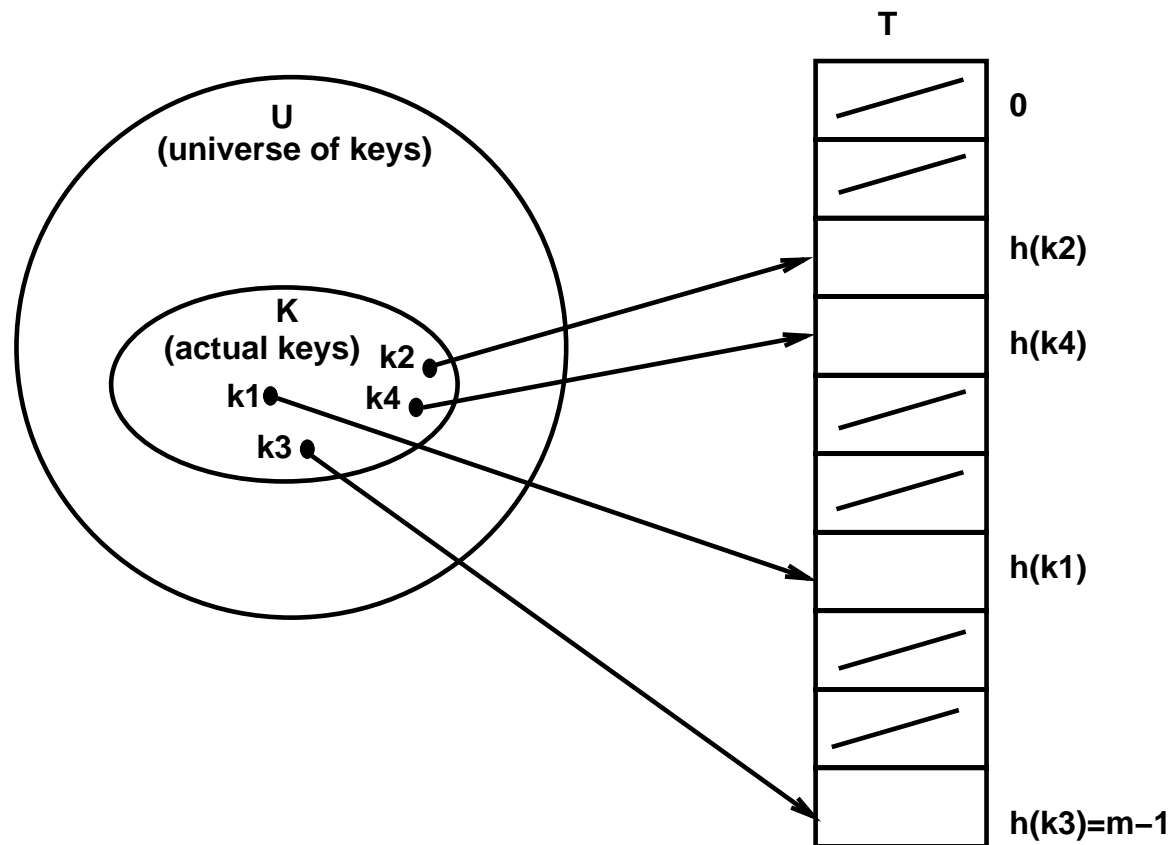
The idea is to store keys in a large hash table of size  $m$  (say  $n \leq m \leq 2n$ , where  $n$  is the number of keys to store) such that the data is scattered across the table uniformly.

The INSERT, SEARCH, DELETE operations on a hash table take  $O(1)$  average case time, as we shall see.

**Example:** A symbol table in a compiler is often represented using a hash table, where each element key consists of the string of characters to store in the table.

## Hash Tables *continued*

The element with key  $k$  is stored in slot  $h(k)$  given that  $h$  is a function which maps a key  $k \in U$  to an index of hash table  $T[0..m-1]$ . Note that  $h : U \rightarrow \{0, 1, \dots, m-1\}$  given that  $m$  is the size of the hash table.  $h$  should be computable in  $O(1)$  time.



## Hash Tables *continued*

---

If  $K$  represents the set of keys to store and  $|K| \ll |U|$ , then, by using a hash table, we can reduce the storage space for  $T$  to  $\Theta(|K|)$  and still access the elements in  $O(1)$  average case time.

Hashing schemes perform very well in practice when the maximum number of keys is known in advance and can be implemented in fewer than 200 lines of C or C++ code!

**Problem:** A **collision** occurs if  $h(k_1) = h(k_2)$ .

It is best to avoid collisions altogether by obtaining a suitable hash function that minimizes the number of collisions. However, if  $|K| > m$ , there will be keys that hash to the same index; hence, collision avoidance is not possible.

Fortunately, there are effective techniques for resolving the conflicts created by collisions.

## Collision Resolution Methods

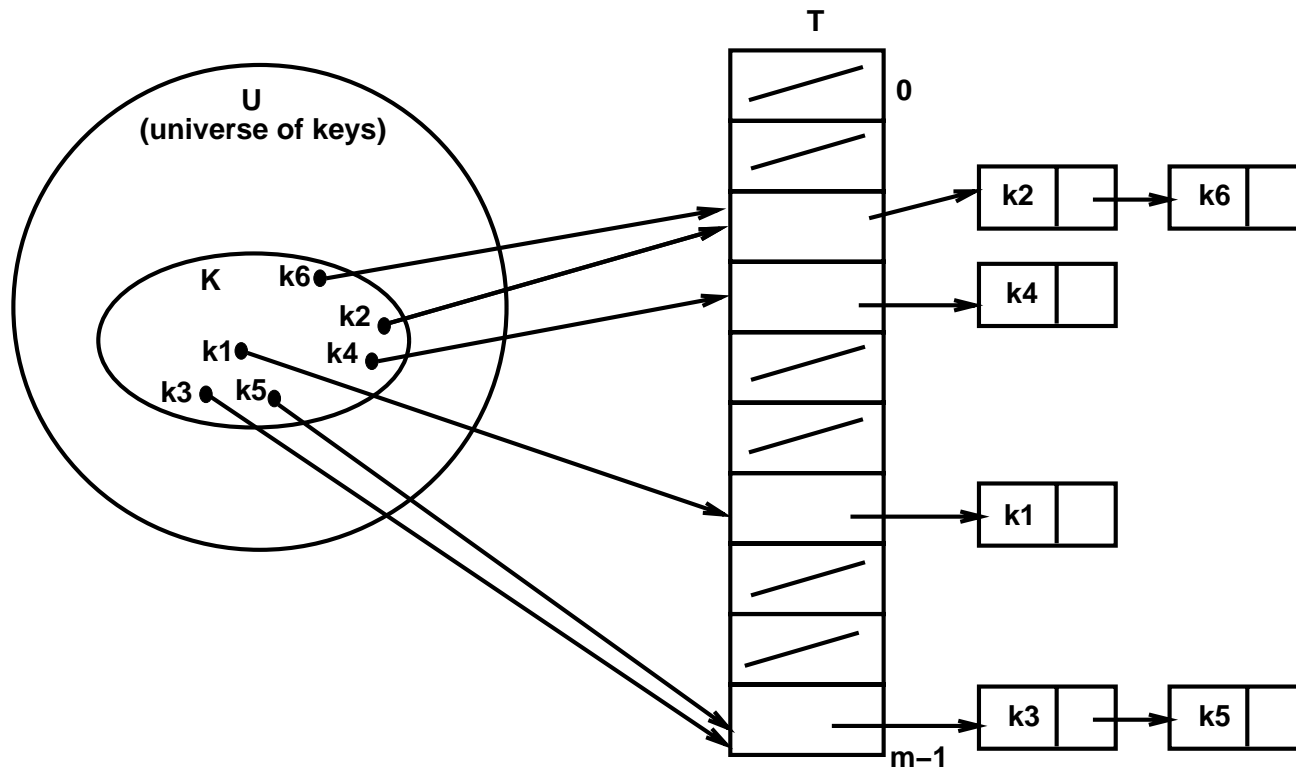
---

There are two methods for resolving collisions:

1. **Chaining:** Keep a linked list of the elements that hash to the same index. This method is simple to implement and degrades gracefully when the number of keys  $n$  exceeds the table size  $m$ . It uses dynamic memory allocation.
2. **Open Addressing:** Store all keys in the table itself, and if a collision occurs, then use some method to obtain another location to place the item in the table. This is easy to implement and no dynamic memory allocation is used; however,  $n$  must be less than or equal to  $m$ , performance degrades as  $n \rightarrow m$ , and deletion operations are difficult to support, as we shall see.

## Collision Resolution by Chaining

The simplest collision resolution technique is called **chaining**. The basic idea is that all items that hash to the same index are stored in a linked list pointed at by that table entry.



## Collision Resolution by Chaining *continued*

---

The operations on the chained hash table are defined below:

1. CHAINED-HASH-SEARCH( $T, k$ )  
search for an element with key  $k$  in  $T[h(k)]$

The worst case running time is proportional to the length of the list.

2. CHAINED-HASH-INSERT( $T, x$ )  
insert  $x$  at the head of the list  $T[h(key[x])]$

The worst case running time is  $O(1)$ .

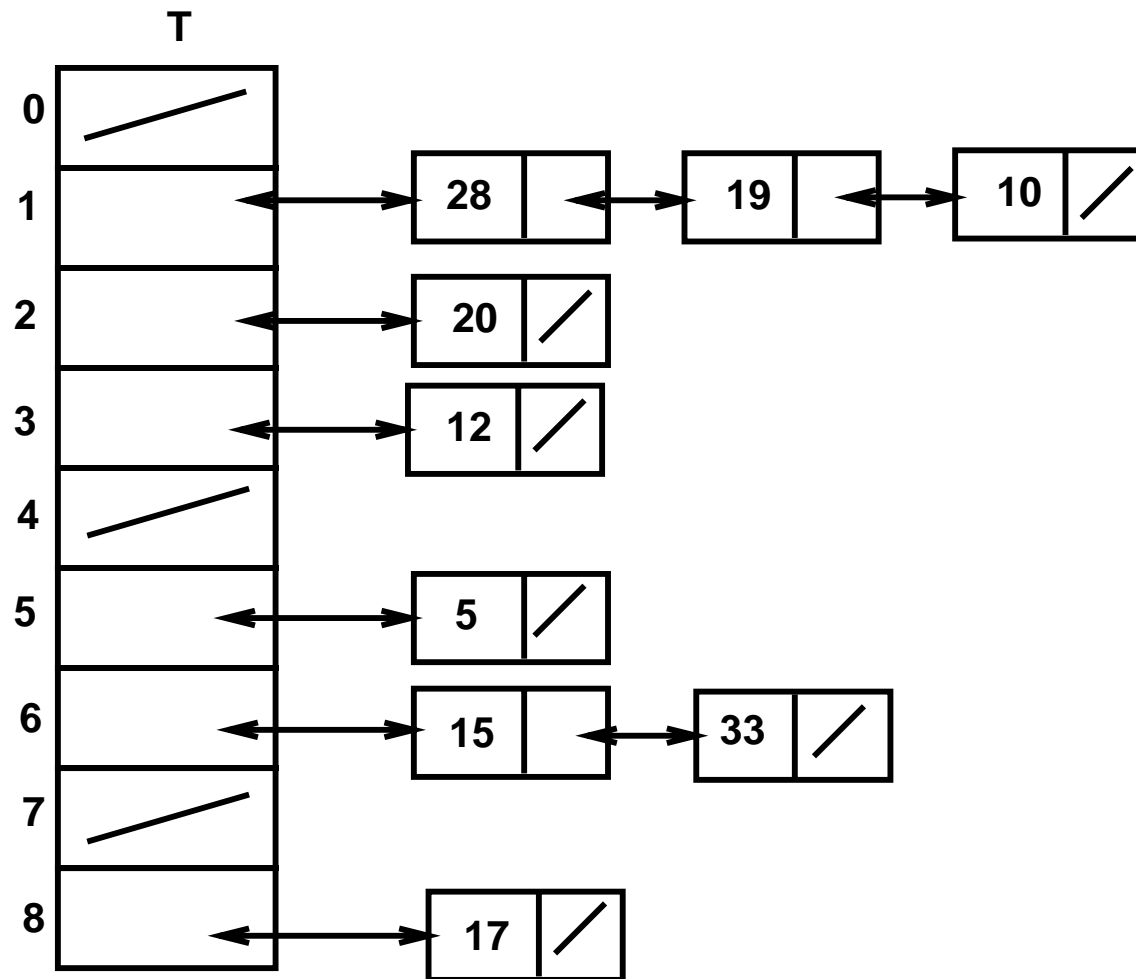
3. CHAINED-HASH-DELETE( $T, x$ )  
delete  $x$  from list  $T[h(key[x])]$

The worst case running time is  $O(1)$  if the list is doubly linked. If singly linked, it must search for  $x$ 's predecessor to perform the operation.

## An Example of Hashing with Chaining

---

**Example:**  $m = 9, h(k) = k \bmod 9, K = \{5, 10, 19, 33, 20, 15, 12, 17, 28\}$



## Analysis of Hashing with Chaining

---

The worst case running time to search for an element in hashing with chaining is  $\Theta(n)$ , when all keys hash to the same slot.

The average case depends on how well the  $n$  keys are distributed among the  $m$  slots.

Given a hash table of size  $m$  and  $n$  elements to store in the table, we define **load factor** as  $\alpha = \frac{n}{m}$ . Our analysis will be in terms of  $\alpha$ .

For analysis, we will make the assumption of **simple uniform hashing**, which means that any given element is equally likely to hash into any of the hash table slots, independently of the other elements.



## Analysis of Hashing with Chaining *continued*

---

**Theorem:** In a hash table in which collisions are resolved by chaining, an unsuccessful search takes time  $\Theta(1 + \alpha)$ , on average, under the assumption of simple uniform hashing.

**Proof:**

- The assumption of simple uniform hashing implies that any key  $k$  is equally likely to hash to any of the  $m$  slots.
- The average time to search unsuccessfully for key  $k$  is the average time to search one of the  $m$  lists.
- Given that the average length of a list is  $\alpha = \frac{n}{m}$ , the expected number of elements to examine in an unsuccessful search is  $\alpha$ .
- Hence, the total time required is  $\Theta(1 + \alpha)$  (including the time to compute  $h(k)$ ).

## Analysis of Hashing with Chaining *continued*

---

**Theorem:** In a hash table in which collisions are resolved by chaining, a successful search takes time  $\Theta(1 + \alpha)$ , on average, under the assumption of simple uniform hashing.

**Proof:** See the textbook.

## Hash Table Size

---

The following are guidelines on the selection of hash table size,  $m$ :

- Select  $m$  as a prime number or power of two. In theory the prime is slightly better (more choice when picking a hash function and universal hashing can be used more easily), but in practice we tend to choose  $m = 2^k$  (eliminates the need to do modulo operations).
- If using chaining, use  $n \leq m \leq 2n_{max}$ , where  $n_{max}$  is the maximum number of keys to be stored in the table at one time. Nothing hinges on a particular choice.
- For open addressing with double hashing, use  $\frac{3}{2}n_{max} \leq m \leq 2n_{max}$ . Larger values result in better performance.

## Hash Functions

---

Often keys are similar or clustered, but we do not want this regularity of key distribution to affect uniformity of the hash function.  $h(x)$  and  $h(x + \varepsilon)$  should differ, where  $\varepsilon$  represents some small change (e.g., integer, real, or string).

A good hash function should come close to satisfying the assumption of simple uniform hashing. More formally, assume that each key is drawn independently from  $U$  according to the probability distribution  $P$ , where  $P(k)$  is the probability that key  $k$  is drawn, then:

$$\sum_{k:h(k)=j} P(k) = \frac{1}{m} \quad j = 0, 1, \dots, m-1$$

Unfortunately,  $P$  is generally unknown (though sometimes it is, for example, random reals in the range  $[0..1)$ ).

Usually heuristic techniques are used to create hash functions that perform fairly well. Sometimes stronger assumptions are required than simple uniform hashing.

Most hash functions assume that the universe of keys is the set of natural numbers (or that they can be mapped to natural numbers).

## Hash Functions *continued*

---

**Division method:**  $h(k) = k \bmod m$

**Example:** If  $m = 52$  and  $k = 235$ , then  $h(k) = 27$ .

**Rules of thumb:**

- Avoid powers of 2 as the value of  $m$ ; otherwise, not all bits of the key will be used by the function. If  $m = 2^p$ , then only the lowest order  $p$  bits will be used!
- Avoid powers of 10 when decimal numbers are used as keys.
- If  $m = 2^p - 1$  and  $k$  is a character string interpreted in radix  $2^p$ , then keys that are permutations of the same digits hash to the same slot! For example, 51 in radix  $2^4$  ( $5 \times 16^1 + 1 \times 16^0 \bmod 15 = 6$ ) collides with 15 in radix  $2^4$  ( $1 \times 16^1 + 5 \times 16^0 \bmod 15 = 6$ ) when  $m = 15$ . (See 11.3-3.)
- Good values of  $m$  are primes not too close to a power of 2.

**Example:** If  $n = 2000$ , and we are willing to examine 4 elements on average during search, then  $h(k) = k \bmod 491$  is a good choice (512 is the power of 2).

## **Hash Functions** *continued*

---

### **Multiplication method:**

Refer to the book for further details on this method.

## Open Addressing

---

**Open Addressing:** All elements are stored directly in the hash table (i.e., there are no pointers); hence, either there is an element or NIL in a table entry.

The hash table is now a static entity that can fill up, so  $\alpha \leq 1$ . This method trades pointers for table size.

To perform a table insertion, we now **probe** the hash table for an empty slot in some systematic way. Instead of using a fixed order; however, the sequence of positions probed depends on the key to be inserted.

The hash function is redefined as:

$$h : U \times \{0, 1, \dots, m-1\} \rightarrow \{0, 1, \dots, m-1\}$$

For every key  $k$ , the **probe sequence**  $\langle h(k, 0), h(k, 1), \dots, h(k, m-1) \rangle$  is considered. As the table fills up, every position in the table is a possible option for the insertion of a key.

## Open Addressing *continued*

---

We assume that we are storing only keys in the table; there is no satellite data; hence, a table slot either contains a key  $k$  or NIL.

```
HASH-INSERT( $T, k$ )
1.  $i \leftarrow 0$ 
2. repeat  $j \leftarrow h(k, i)$ 
3.         if  $T[j] = \text{NIL}$ 
4.             then  $T[j] \leftarrow k$ 
5.                 return  $j$ 
6.             else  $i \leftarrow i + 1$ 
7. until  $i = m$ 
8. error “hash table overflow”
```



## Open Addressing *continued*

---

HASH-SEARCH uses the hash function to search for key  $k$ , and it will terminate if it finds  $k$  in the slot, returning the index where the item is found. It also terminates with NIL if it finds NIL in a probed slot or if it searches the entire table without success.

HASH-SEARCH( $T, k$ )

1.  $i \leftarrow 0$
2. **repeat**  $j \leftarrow h(k, i)$
3.       **if**  $T[j] = k$
4.       **then return**  $j$
6.        $i \leftarrow i + 1$
7.   **until**  $T[j] = \text{NIL}$  or  $i = m$
8. **return** NIL

## Deleting in an Open-address Table

---

We cannot simply delete an element in a table by inserting a NIL; it could break the probe sequence for other elements in the table.

It is possible to get around this by using pointers, which goes against the spirit of open addressing.

We could also use a special purpose delete marker instead of NIL when an element is removed.

Then HASH-SEARCH will automatically search further when looking for other elements when encountering the delete marker. HASH-INSERT can be modified to treat the delete marker as NIL. However, using this approach, the search time is no longer dependent only on the load factor  $\alpha$ .

Because deletion in open-address hash tables is difficult, usually open-address hashing is not done when delete operations are required.

## Probe Sequences

---

In the analysis of open addressing, we make the assumption of **uniform hashing**, which requires that each key considered is equally likely to have any of the  $m!$  permutations of  $\{0, 1, \dots, m-1\}$  as its probe sequence.

Three techniques are commonly used to compute probe sequences for open addressing:

1. linear probing
2. quadratic probing
3. double hashing

These techniques guarantee that  $\langle h(k, 0), h(k, 1), \dots, h(k, m-1) \rangle$  is a permutation of  $\langle 0, 1, \dots, m-1 \rangle$  for each key  $k$ , but none fulfills the assumption of uniform hashing since none can generate more than  $m^2$  sequences.

## Linear Probing

---

Given  $h' : U \rightarrow \{0, 1, \dots, m-1\}$ , **linear probing** uses the hash function:  $h(k, i) = (h'(k) + i) \bmod m$  for  $i = 0, 1, \dots, m-1$ .

Given key  $k$ , the first slot probed is  $T[h'(k)]$ , then  $T[h'(k) + 1]$ ,  $T[h'(k) + 2]$ , etc. Hence, the first probe determines the remaining probe sequence.

**Example:**  $h'(k_1) = h'(k_2) = j$ ,  $h'(k_3) = j + 1$ ,  $h'(k_4) = j$ , and the keys are entered in the order:  $k_1, k_2, k_3, k_4$ , giving the following table:

--	--	--	--	--	--	--	--

This method is easy to implement, but suffers from **primary clustering**, that is, two keys that hash to different locations compete with each other for successive rehashes. Hence, long runs of occupied slots build up, increasing search times.

## Quadratic probing

---

**Quadratic hashing** uses a hash function of the form:

$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$$

where  $h'$  is an auxiliary hash function,  $c_1, c_2 \neq 0$  are auxiliary constants, and  $i = 0, 1, \dots, m - 1$ . Note that  $m$ ,  $c_1$ , and  $c_2$  must be selected carefully.

The initial probe position is  $T[h'(k)]$ , with subsequent positions depending on a quadratic function of the probe number  $i$ .

**Example:** probe sequence given  $c_1 = c_2 = 1$ :

	$p_1$		$p_2$				$p_3$						$p_4$	
--	-------	--	-------	--	--	--	-------	--	--	--	--	--	-------	--

Quadratic probing is better than linear probing because it spreads subsequent probes out from the initial probe position. However, when two keys have the same initial probe position, their probe sequences are the same, a phenomenon known as **secondary clustering**.

## Double Hashing

---

**Double hashing** is one of the best open addressing methods available because the permutations produced have many of the characteristics of randomly chosen permutations. It uses a hash function of the form:

$$h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m$$

where  $h_1$  and  $h_2$  are auxiliary hash functions.

The initial position probed is  $T[h_1(k)]$ , with successive positions offset by the amount  $(i h_2(k))$  modulo  $m$ .

Now, keys with the same initial probe position can have different probe sequences.

Note that  $h_2(k)$  must be relatively prime to  $m$  for the entire hash table to be accessible for insertion and search. If  $d = \text{GCD}(h_2(k), m) > 1$  for some key  $k$ , then search for key  $k$  would only access  $\frac{1}{d}$ th of the table. (See Chapter 33.)

## Double Hashing *continued*

---

A convenient way to ensure that  $h_2(k)$  is relatively prime of  $m$  is to select  $m$  as a power of 2 and design  $h_2$  to produce an odd positive integer. Or, select a prime  $m$  and  $h_2$  to produce a positive integer less than  $m$ .

**Example:**  $h_1(k) = k \bmod m$ ,  $h_2(k) = 1 + (k \bmod m')$ , where  $m'$  is slightly less than  $m$ .

Double hashing is an improvement over linear and quadratic probing in that  $\Theta(m^2)$  sequences are used rather than  $\Theta(m)$ , since every  $(h_1(k), h_2(k))$  pair yields a distinct probe sequence, and the initial probe position,  $h_1(k)$ , and offset,  $h_2(k)$ , vary independently.

## Double Hashing *continued*

---

**Example:** Use double hashing to store the keys 10, 18, and 34 in a hash table of size  $m = 8$ , using:

$$h_1(k) = k \bmod 8, h_2(k) = 1 + (k \bmod 6)$$

--	--	--	--	--	--	--	--



## Analysis of Open-Address Hashing

---

**Theorem:** Given an open-address hash table with load factor  $\alpha = \frac{n}{m} < 1$ , the expected number of probes in an unsuccessful search is at most  $\frac{1}{1-\alpha}$ , assuming uniform hashing.

**Proof:**

- Refer to the book for the proof.

## Analysis of Open-Address Hashing *continued*

---

**Corollary 12.6.** Inserting an element into an open-address hash table with load factor  $\alpha$  requires at most  $\frac{1}{1-\alpha}$  probes on average, assuming uniform hashing.

**Proof:**

Insertion of a key requires an unsuccessful search followed by the placement of the key in the first empty slot found. Thus the expected number of probes is at most  $\frac{1}{1-\alpha}$ .

## Analysis of Open-Address Hashing *continued*

---

**Theorem:** Given an open-address hash table with load factor  $\alpha < 1$ , the expected number of probes in a successful search is at most  $\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$ , assuming uniform hashing and assuming that each key in the table is equally likely to be searched for.

**Proof:** Refer to the book for the proof.