

Chapter 4

Hashing

“The key is the address”

The Dictionary ADT

A *dictionary* is a set data type that supports 3 operations: *INSERT*, *DELETE*, and *SEARCH*.

- Which operation do you think is used most often?

A *linked list* or an *array* can be used to implement a dictionary.

However, searching is not very efficient ($O(n)$ time where n is the number of elements in the set).

Hashing

Idea: *hashing* consists of 2 parts.

- A *table* (an array) of size m ($T[0..m-1]$);
 - Here, m is a number picked by the designer.
- A *hash function* h which maps a key (of an element) into an index (integer) between 0 and $m-1$ inclusive.

An element with key k is mapped to slot $h(k)$ of the array.

Hashing

Example: suppose we want to keep one record per student in Class 2119. The key of each student record is his/her university number.

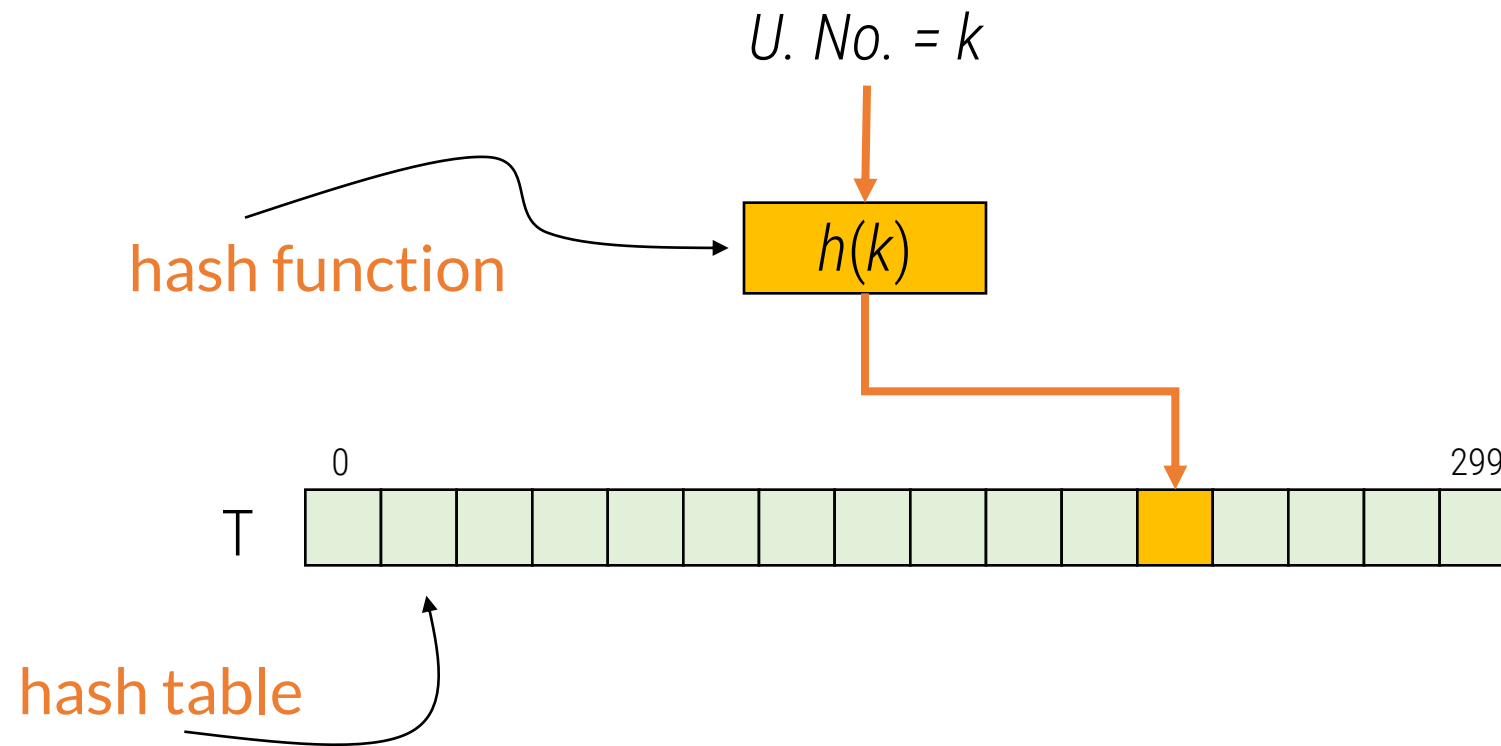
We may use a hash table with 300 entries/slots ($T[0..299]$) to store those records in the following way:

- Interpret a University Number as an integer
- Each student is mapped into a slot by computing the hash function:

$$h(\text{U.No.}) = (\text{U.No.}) \bmod 300$$

A hash value would range from 0 to 299.


Example



Two problems

Example: what if we take student names as keys?

We can interpret a student's name as an integer expressed in a suitable radix notation.

- e.g., ASCII uses an integer code per character. There are in total 128 characters in the ASCII set. The string "BEN" can be represented by 3 codes: "66", "69", "78".
- $h(\text{"BEN"}) = [66 \times (128)^2 + 69 \times 128 + 78] \bmod 300$
= 54

- $h(\text{"ALAN"}) = [65 \times (128)^3 + 76 \times (128)^2 + 65 \times (128) + 78] \bmod 300$
= 162

Problem with Hashing

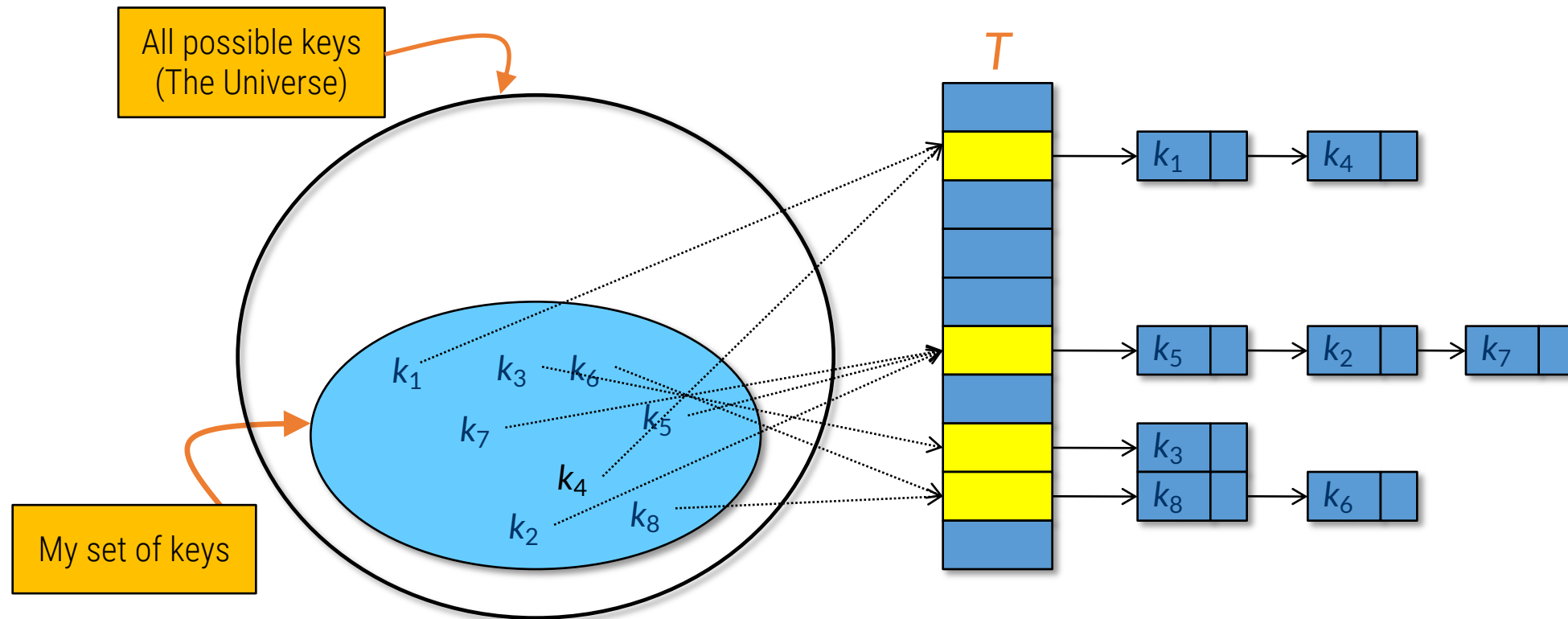
2 keys may have the same hash value — a *collision*.

To handle collision, we need:

- a good hash function, such that collision does not occur often,
- a collision resolution strategy.

Chaining (a.k.a open hashing)

Keep a linked list of all elements that have the same hash value.



Chaining (implementation)

```
typedef struct node *node_ptr;  
struct node {  
    element ele;  
    node_ptr next;  
};  
  
typedef node_ptr LIST;  
  
LIST T[m];
```

To initialize the hash table

```
Chained_Hash_Init(T) {  
    int i;  
    for (i=0;i<m;i++)  
        T[i] = NULL;  
}
```

Chaining (implementation)

To search an element with key = k

```
Chained_Hash_search(T,k) {  
    node_ptr p;  
    p = T[h(k)];  
    while ((p != NULL) &&  
           (p->ele.key != k))  
        p = p->next;  
    return(p);  
}
```

To insert an element x

```
Chained_Hash_Insert(T,x) {  
    node_ptr p;  
    p = T[h(x.key)];  
    insert x into the linked list pointed to by p;  
  
    /* note: we might want to perform duplicate check */  
}
```

Chaining (implementation)

For delete, if we only have the key of the element to be deleted, we have to search for it first.

note: all the variations of linked list may be applied (although we usually use simple singly linked list, why?).

Chaining (analysis)

Load Factor (a)

Given a hash table T with m slots that stores n elements, we define the load factor a for T as n/m . Intuitively, a is the *average number of nodes (elements) stored in a chain*.

Insert is easy, it takes constant time $O(1)$.

Delete: constant time if we have the pointer to the node and the list is a doubly linked list. Otherwise, its running time is similar to searching.

Chaining (analysis)

Searching: the worst case happens when all the keys stored are mapped to the same slot.

- e.g., if we use the following hash function:
- $h(\text{U.No.}) = (\text{U.No.}) / 100000000 \bmod 300$
most of our class members will be mapped to 3.

Integer division



What is the search complexity in the worst case?

- $O(n)$ (no better than linked list).

Simple uniform hashing

The performance of hashing depends very much on the hash functions. In general, a good hash function should **map the keys as uniformly as possible over the m slots**, i.e., the m slots should be given roughly the same number of elements.

We call this ideal hash function *simple uniform hashing*.

Search complexity

The average running time of search would be the time required to evaluate the hash function + the time to traverse the list.

In an *unsuccessful search*, the number of elements examined is *a on average*. Therefore,

- average complexity of unsuccessful search is $O(1+a)$.
- Q: Why $O(1+a)$ and not simply $O(a)$?
- Q: What about successful search?

Search complexity

In a *successful search*, the average number of elements examined is about $1 + a/2$.

Average complexity of successful search = $O(1+1+a/2) = O(1+a)$.

Hash functions

A good hash function should

- satisfy (approximately) the assumption of simple uniform hashing: *if you randomly draw a key from the universe, it is equally likely that the key get mapped to any of the m slots,*
- be easy to compute (as efficient as possible).

Hash functions

If the universe of keys is not the set of natural numbers (i.e., $0, 1, 2, \dots$), we need to convert the keys to natural numbers first (e.g., by transforming a string to a number using the ASCII code).

In the following discussion, we assume the keys have been transformed to natural numbers and only consider ways of mapping natural numbers to $[0..m-1]$.

The division method

For a key k and a table size m :

- $h(k) = k \bmod m$

With the division method, the choice of m is *very important*.

The division method

Example: table size = 100.

Key	ASCII	Address
TO	124117	17
IT	111124	24
IS	111123	23
AS	101123	23
AT	101124	24
IF	111106	06
OF	117106	06
AM	101115	15
BE	102105	05
DO	104117	17
AN	101116	16
GO	107117	17
SO	124117	17

$$h(c_1c_2) = [\text{ASCII}(c_1) * 1000 + \text{ASCII}(c_2)] \bmod 100$$

05	BE
06	IF, OF
07	
...	...
10	
11	
...	...
14	
15	AM
16	AN
17	TO, DO, GO, SO
...	...
23	AS, IS
24	IT, AT
25	

Obs: only the last character matters.

Q: how many slots (out of the 100) are ever used?

The division method

A good hash function should use *all the information* provided by the keys.

Some general rules of thumb for choosing the value of m (for the division method).

1. m should not be a power of 2. Reason: if $m = 2^p$, then $h(k)$ is just the p lowest order bits of k .
2. m should not be a power of 10 if the application deals with decimal numbers.
3. Good values of m are primes.
4. If (3) is too difficult to meet, pick m such that it has no prime factors less than 20 (e.g., 23×31).

Division method

Example: design a hash table to hold roughly $n = 2000$ elements.
Suppose we don't mind searching a linked list of size 3 (on average, for an unsuccessful search)

- $\Rightarrow a = n/m = 3$
- $\Rightarrow m = 2000/3 = 667$
- 667 is not a prime ($667 = 23 \times 29$)
- We should find a prime close to 667 but not close to any powers of 2 (i.e., 512, 1024).
- Let's pick 701.
- $h(k) = k \bmod 701$.

Open addressing (a.k.a. closed hashing)

In **open addressing**, all elements are stored in the hash table itself. That is, each table entry contains either an element of the dictionary or a special value (called “NIL”) which signifies that no element is stored in the slot.

element $T[m]$;

NIL can be represented by an invalid key value, e.g., “0000000000” as an invalid U. No.

Open addressing

Compared with chaining, open addressing avoids pointers.

TWO advantages:

- it saves time in not doing memory allocation and de-allocation,
- it saves space
 - Pointers take space too!
 - 4 bytes for 32-bit machines
 - 8 bytes for 64-bit machines

Space efficiency

If we are given the same amount of memory, we can afford a larger number of slots under open addressing. This potentially yields fewer collisions.

Space efficiency

Example

- Chaining:
 - Say each element takes $x = 10$ bytes
 - n elements stored in a table of m entries
 - Say load factor $a = n/m = 3$
 - Array T : $4 * m$; each element: $10 + 4$ (assuming 4 bytes per pointer)
 - Total amount of memory = $46m$
- Open addressing
 - $46m$ bytes of memory \Rightarrow a table of $4.6m$ entries under open addressing.

Collision resolution with open addressing

When we are inserting an element, if a collision occurs, alternative slots are tried until an empty slot is found (this process is called *probing*).

Formally, we consider a *probe sequence*:

$\langle h(k, 0), h(k, 1), \dots, h(k, m-1) \rangle$ where

$$h(k, i) = [h_1(k) + f(i)] \bmod m$$

for some function $f(\cdot)$ of which $f(0) = 0$.

$h_1(\cdot)$: hash function $f(\cdot)$: increment function

Constraint: the probe sequence should be a permutation of $\langle 0, 1, \dots, m-1 \rangle$ so that every slot is included in the sequence.

Insertion

```
Open_Address_Hash_Insert(T,x) {  
    i = 0; /* collision count */  
    k = x.key;  
    do {  
        j = h(k,i);  
        if (T[j].key == NIL) {  
            copy x to T[j]; return; }  
        else  
            i++;  
    } while (i < m)  
    return("hash table overflow!");}
```

A hash table for “singles”

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

$$h(k,i) = [h_1(k)+i] \bmod 10$$

Suppose:

$$h_1(\text{“Anita”}) = 2$$

$$h_1(\text{“John”}) = 5$$

$$h_1(\text{“Paul”}) = 7$$

$$h_1(\text{“Ben”}) = 2$$

$$h_1(\text{“Mary”}) = 1$$

$$h_1(\text{“Catherine”}) = 1$$

A hash table for “singles”

0	
1	Mary
2	Anita
3	Ben
4	Catherine
5	John
6	
7	Paul
8	
9	

$$h(k,i) = [h_1(k)+i] \bmod 10$$

Suppose:

$$h_1(\text{“Anita”}) = 2$$

$$h_1(\text{“John”}) = 5$$

$$h_1(\text{“Paul”}) = 7$$

$$h_1(\text{“Ben”}) = 2$$

$$h_1(\text{“Mary”}) = 1$$

$$h_1(\text{“Catherine”}) = 1$$

Note: “Catherine” is *moved* from slot 1 to 4 following a “chain” of elements.

Search

To search an element, we follow the same probe sequence until the element is found or when an empty slot is encountered.

```
Open_Address_Hash_Search(T,k) {  
    i = 0;  
    do {  
        j = h(k,i);  
        if (T[j].key == k)  
            return(j);  
        i++;  
    } while ((T[j].key != NIL) && (i<m))  
    return("Not found!");}
```

A hash table for “singles”

0	
1	Mary
2	Anita
3	Ben
4	Catherine
5	John
6	
7	Paul
8	
9	

Suppose $h_1(\text{“Peter”}) = 5$.
Want to search for “Peter”.

Delete

To delete an element, we follow the same probe sequence until the element is found, or NIL is encountered (i.e., the element is not in there).

Q: Can we just delete the element from the table? (i.e., put “NIL” in the slot?)

A: NO!

A hash table for “singles”

0	
1	Mary
2	Anita
3	
4	Catherine
5	John
6	
7	Paul
8	
9	

- E.g., “Ben” just got married (delete “Ben”), and someone is interested in “Catherine” and would like to see if she is available (search “Catherine”).
- The “chain” which leads to “Catherine” is broken!
- Remedy: do not break the chain but maintain that the slot which “Ben” used to occupy is now available for insertion.
- To do so, we mark the slot by storing in it the special value “DELETED” instead of “NIL”

A hash table for “singles”

0	
1	Mary
2	Anita
3	<i>Deleted</i>
4	Catherine
5	John
6	
7	Paul
8	
9	

- E.g., “Ben” just got married (delete “Ben”), and someone is interested in “Catherine” and would like to see if she is available (search “Catherine”).
- The “chain” which leads to “Catherine” is broken!
- Remedy: do not break the chain but maintain that the slot which “Ben” used to occupy is now available for insertion.
- To do so, we mark the slot by storing in it the special value “DELETED” instead of “NIL”

Deletion

```
Open_Address_Hash_Delete(T,k) {  
    i = 0;  
    do {  
        j = h(k,i);  
        if (T[j].key == k) {  
            T[j].key = DELETED;  
            return;}  
        i++;  
    }  
    while ((T[j].key != NIL) && (i<m))  
    return("Not Found!");}
```

Q: Do we have to modify
Open_Address_Hash_Search()?

Deletion

Insert would have to be changed so that it is looking for an empty slot (“NIL”) or a deleted slot (“DELETED”)

```
Open_Address_Hash_Insert(T,x) {  
    .....  
    if ((T[j].key == NIL) || (T[j].key == DELETED)) {  
        .....  
    }  
}
```

Load factor

Because all the data goes inside the table, a bigger table is needed for open addressing than for chaining.

In particular, the load factor α in chaining can be > 1 but for open addressing, it should be less than 1.

Generally, the load factor for open addressing *should be below 0.5*.

Probe sequence

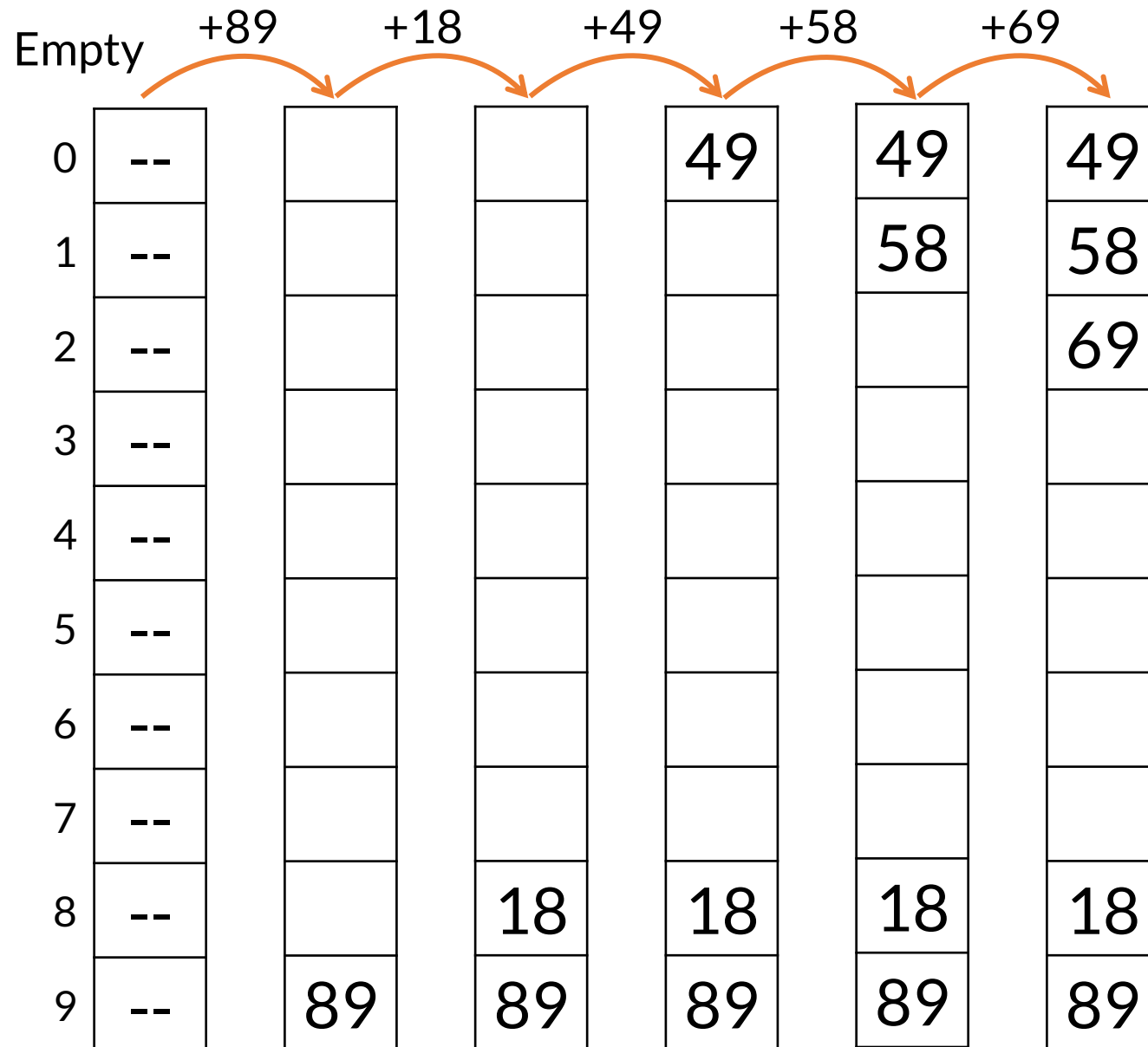
Depending on the increment function $f(i)$ as in

$$h(k,i) = [h_1(k) + f(i)] \bmod m$$

we have different types of probe sequences.

Linear Probing

For linear probing, $f()$ is a linear function of i , typically, $f(i) = i$. This amounts to trying the slots sequentially (with wrap around) in search of an empty slot.



Closed hash table
with linear probing

$$h_1(x) = x \bmod 10$$

$$f(i) = i$$

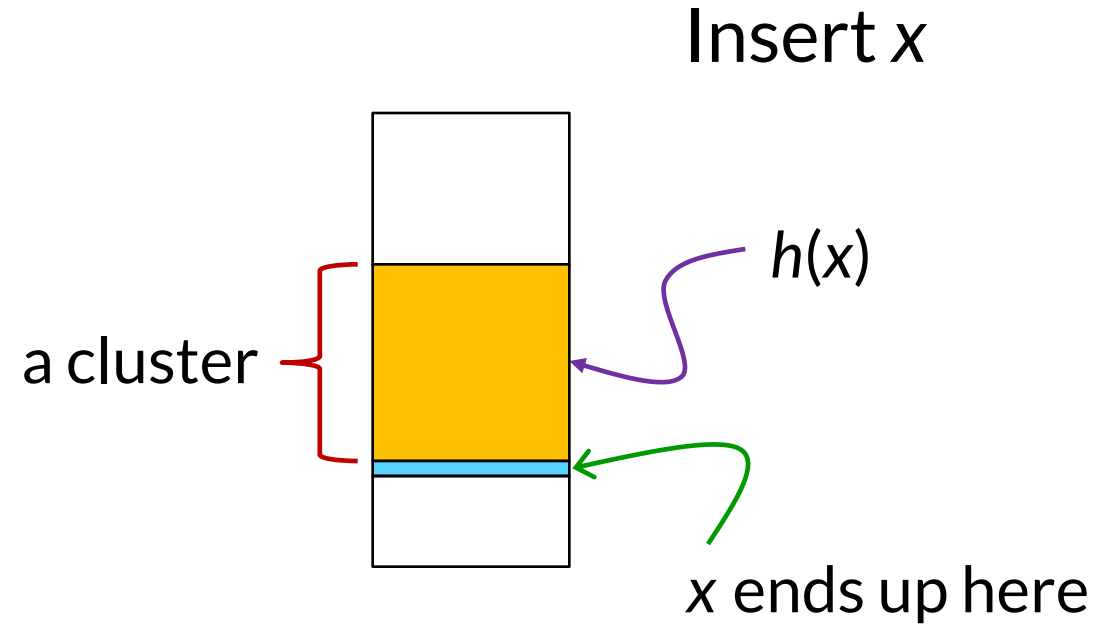
Primary clustering

Problem with linear probing: blocks of consecutively occupied slots occur. These are called *clusters*.

For any key that “hashes” into a cluster, it will require several attempts to resolve collisions, and then the key will add to the cluster.

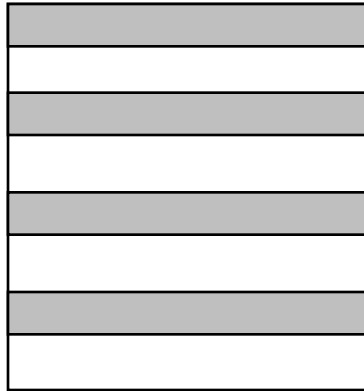
The bigger the cluster is, the faster it grows!

Primary clustering



Primary clustering

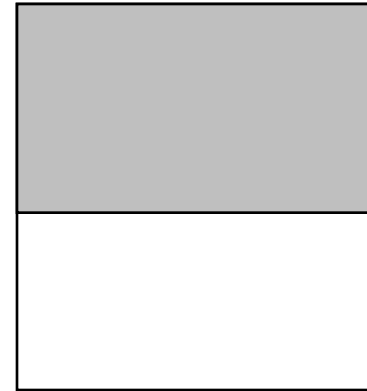
Primary clustering makes searching inefficient



m slots with
 $n = m/2$ slots
occupied

No clustering.

avg. unsuccessful search
 $= (1/2)(2) + (1/2)(1)$ probes
 $= 1.5$ probes



Severe clustering.

$= (1/2)(n/2) + (1/2)(1)$
 $= n/4 + 1/2$
 $\approx m/8$ probes

Primary clustering

Key to the problem of primary clustering: There is only 1 probe sequence for all the keys (although with different starting points.)

Quadratic Probing

Try to eliminate primary clustering by allowing more probe sequences.

For quadratic probing, $f()$ is a quadratic function of i , e.g.,

- $h(k, i) = (h_1(k) + i^2) \bmod m$

	Empty	+89	+18	+49	+58	+69
0	--			49	49	49
1	--					
2	--				58	58
3	--					69
4	--					
5	--					
6	--					
7	--					
8	--		18	18	18	18
9	--	89	89	89	89	89

Closed hash table with
quadratic probing

$h(k,i) = (h_1(k) + i^2) \bmod 10$,
where $h_1(k) = k \bmod 10$.

Quadratic probing

Problem with quadratic probing:

- the probe sequence may not span the whole table;
- there are only m probe sequences (elements that have the same hash value will probe the same slot sequence). This is called *secondary clustering*.

For your interest: if quadratic probing is used, the table size must be a prime number and that the table should never be $>$ half full.

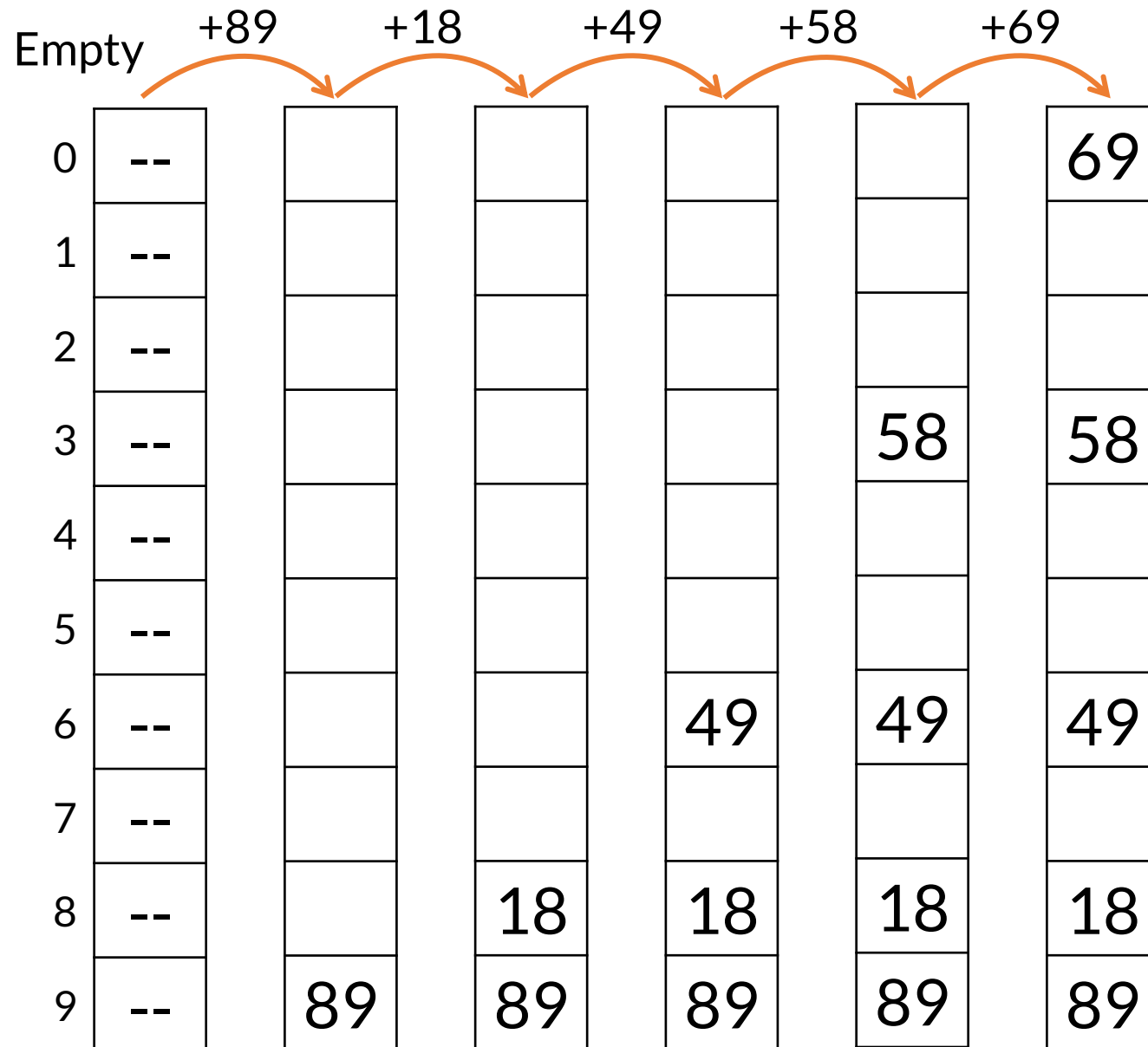
Double Hashing

Double hashing uses *2 hash functions* to define a probe sequence:

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

We use a hash function $h_1(k)$ to determine the initial slot and another $h_2(k)$ to determine the “increment”, or the next slot to probe.

Two keys will have the same probe sequence if they have the same hash values for both $h_1()$ and $h_2()$ (which is unlikely). Clustering therefore does not occur often.



Closed hash table with
double hashing

$$h_1(k) = k \bmod 10,$$

$$h_2(k) = 7 - (k \bmod 7).$$

Double hashing

Some constraints on $h_2(k)$:

- it should never evaluate to 0.
- it should be relatively prime to m (e.g., by setting $m =$ a prime number and $h_2(k)$ to return values $< m$.)

With the above constraints, we can show that all slots in the table will be included in a probe sequence.

Performance analysis

For an open address hash table with load factor $a < 1$

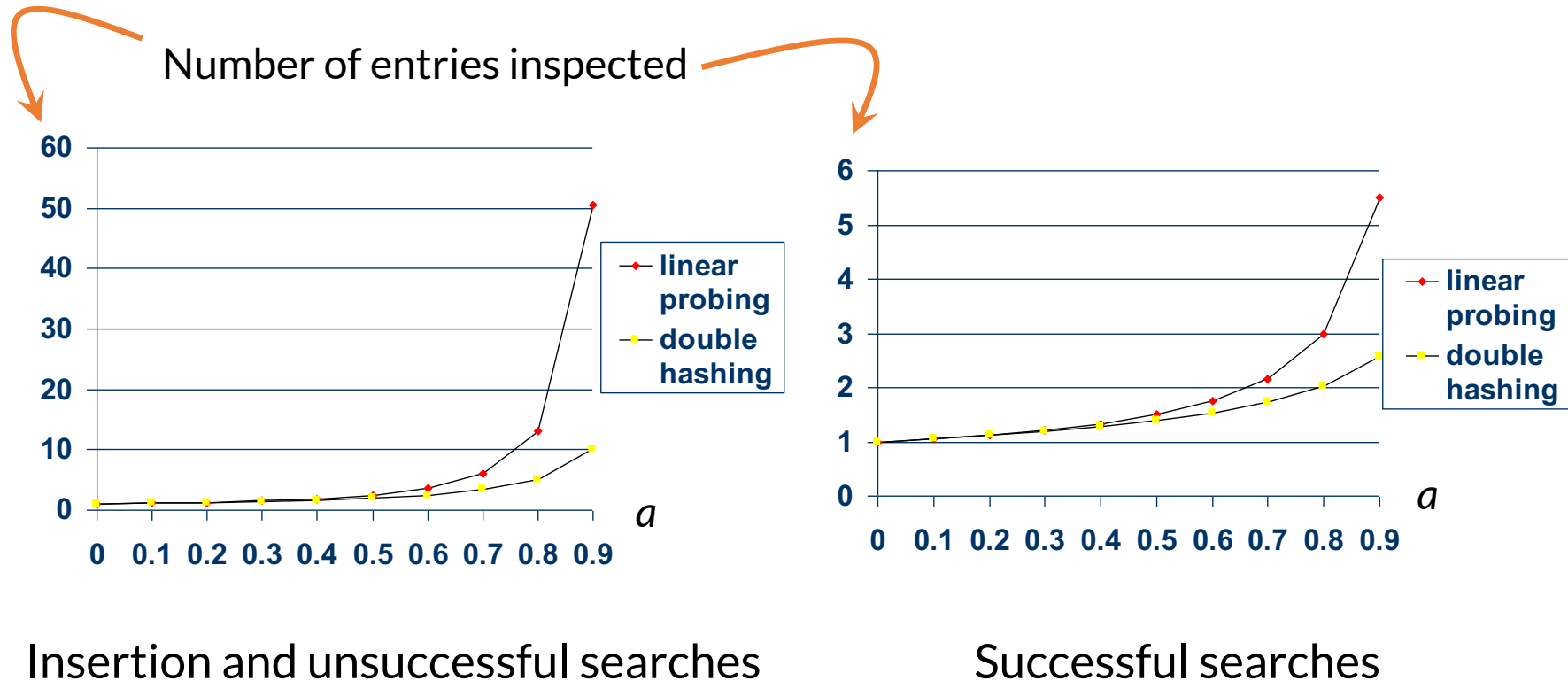
With linear probing, the average number of probes for

- unsuccessful search = $(1 + 1/(1-a)^2) / 2$
- insertion = $(1 + 1/(1-a)^2) / 2$
- successful search = $(1 + 1/(1-a)) / 2$

With double hashing (performance is similar to that of uniform hashing), the average number of probes for

- unsuccessful search = $1/(1-a)$
- insertion = $1/(1-a)$
- successful search = $(1/a)(\ln(1/(1-a)))$

Number of probes vs. load factor



Rehashing

With open addressing, if the table gets too full ($a > 0.5$), the run time of the operations will start taking too long.

Solution: build a larger hash table and rehash the elements to the larger table.

0	6
1	15
2	23
3	24
4	
5	
6	13

rehashing →

0	
1	
2	
3	
4	
5	
6	6
7	23
8	24
9	
10	
11	
12	
13	13
14	
15	15
16	