Outcome (2): A useful data structure - hashing

Application

You need to provide a system to store a set of student records

Name: Peter
Student No.: h02xxxxx
Age: 20
Year: 1
Curriculum: BEng(SE)
Hobbies: working on assignments, programming

Each record can be uniquely identified by a key, denoted by key(x) where x is the record

Functions to be provided:

Insert new records,
delete old records, and
given a key k, return the corresponding record with key = k.

Note: (1) searching is a lot more frequent than the other two operations; (2) the no. of records may be huge.

Another example:
English dictionary

Word: adventure (key)
Pronunciation
Meaning
Sample sentence
adjective / verb etc.

The "Dictionary" ADT

Operations:

Insert(T, x) – insert an element x into a set T
Search(T, k) – search a record with key = k in a set T
Delete(T, x) – delete an element x from a set

The operations look familiar, can we make use of a list implementation ?

Array: insertion/deletion, O(n) too slow
searching O(n) (sorted: O(log n))
Pointer: searching O(n) too slow

Note: search is frequently used

Can we do better?

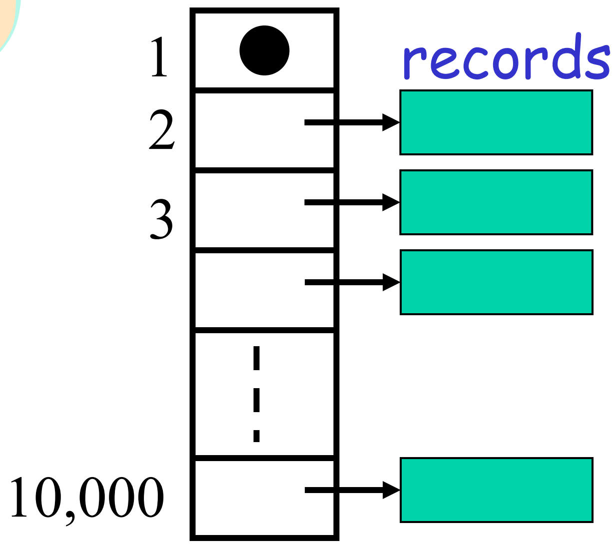
How about this case?

To maintain a set of student records (about 10,000):

Each student is assigned a unique ID from 1 to 10,000.
Searching by ID is the frequent operation.

Q: Any good idea to solve it? [Should be easy!!]

Use an array [1..10,000]



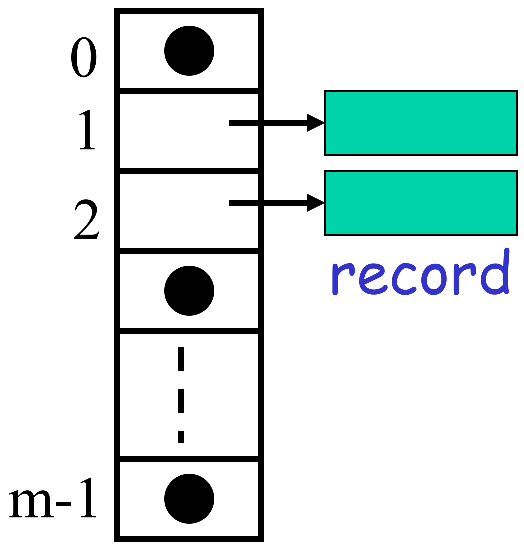
Search, insert, delete can be done in O(1) time!

Direct Addressing

If the universe of possible keys is "integers": {0, 1, ..., m-1}, then

Elements with key i is stored in Table[i]
(or Table[i] can be a pointer to the element)

Direct-address table



Operations: Insert(T, x); Search(T, k); Delete(T, x) [Delete(T, k)]

```
Insert(T, x){
    T[key(x)]= x;
}
```

O(1)

```
Search(T, k){
    return T[k];
}
```

O(1)

```
Delete(T, x){
    T[key(x)]= Nil;}
```

O(1)

Note: We may need to initialize the direct address table.

e.g.

```
for i = 0 to m - 1
    T[i] = nil
```

Time complexity?

O(m), can we do better?

[Learned it in 1st lecture, right?]

So, direct addressing has solved the problem?

e.g. We need to store student records where the format of student id is xxxx – yyyyyy (xxxx: year; yyyyyy: 6 digit random #)

xxxx: 1913 – 2016+; yyyyyy: 000000 - 999999

For direct addressing table; m will be huge; i.e. |U| is large: a naïve implementation => m = 10,000,000,000. But, not all entries have records (each year has about 10000 students)

Two issues in this scheme

(A) Storage: $\Theta(|U|)$ $\begin{cases} |U| \text{ can be very large} \\ |K| \text{ can be relatively small} \end{cases}$

(K: set of keys in dictionary)

Another example:

Consider the words in a dictionary.
Assume that all words are of length <= 20.

Can we use direct addressing to store these records?
Yes, if we have a method to map one word to an index in the array and this mapping can be done in O(1) time.

a $\rightarrow$ 1; b $\rightarrow$ 2; ...; z $\rightarrow$ 26; Convert each word into a radix-26 integer

e.g. cat => $(3)\times 26^2 + 1 \times 26 + 20 \times 1 = 1371$

$|U| = ?$ $\quad 26 \times 26^{19} + 26 \times 26^{18} + \ldots + 26$

$|K| \ll |U|$

Space efficiency problem

(B) For direct-address table to be feasible:

(1) The keys in the universe U can be mapped to I (integer domain)

(2) The mapping $U \rightarrow I$ needs to be one-to-one (1-1).

(3) Given a key k in U, the mapped integer i for k can be computed in O(1) time.

e.g. 'a'-'z'

(1) $a \rightarrow 0$; $b \rightarrow 1$; ...; $z \rightarrow 25$;

(2) true

(3) Yes, ASCII(k) – ASCII('a').

e.g. 100011 – 100211

(1) $100011 \rightarrow 0$; $100012 \rightarrow 1$; ...; $1000211 \rightarrow 200$;

(2) true

(3) Yes, k – 100011.

e.g. {"amy", "eddy",..} How you can do the mapping?

(1) amy $\rightarrow$ 0; eddy $\rightarrow$ 1; ...

(2) true

(3) How can you do it in constant time?

e.g. {“amy”, “eddy,..}

(1) Convert each character into ASCII code and calculate an integer according to the formula:

ASCII(rightmost char) + ASCII(2nd char) x 26 + ASCII(3rd char) x 26^2 + ...

e.g. For “amy”:
ASCII(“y”) + ASCII(“m”) x 26 + ASCII(“a”) x 26^2

(2) true
(3) Yes.

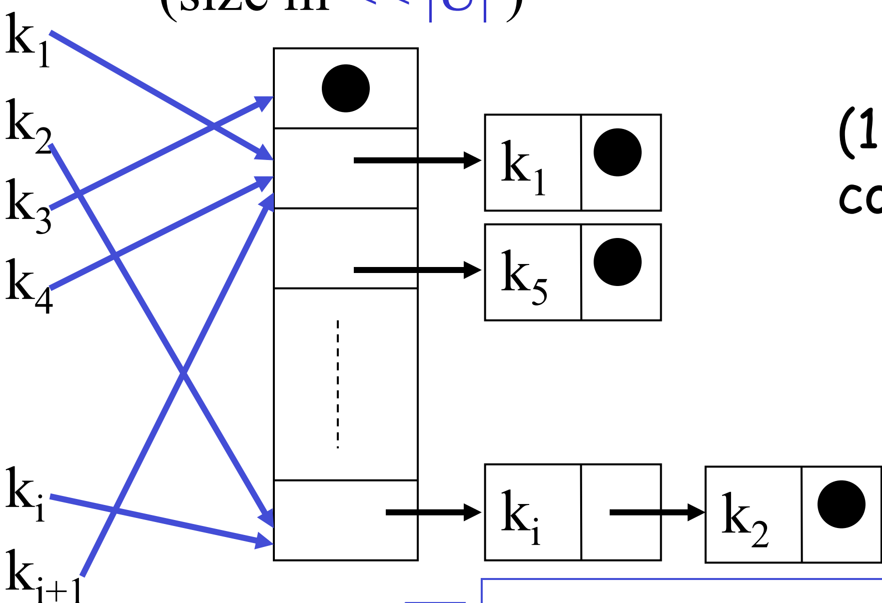
The range of integers will be huge and |K| << |U|.

If $|K| \ll |U|$, can we store data more space efficient while keeping the searching in O(1) time?

Idea: some keys in |U| may not appear, so relax the 1-1 requirement

Note: but this constant time bound is for average case

Hash Table T
(size $m \ll |U|$)



h(k)
hash function
$h: U \rightarrow [0..m-1]$

must be done quickly [O(1)]

Allow more than one key map to the same array entry

Collision: $h(k_i) = h(k_j)$

(1) Try to find a good hash function s.t. collision does not occur that often

What is a good hash function?

(2) Design collision resolution strategy.

Example: chaining
keep a linked list of elements with same hash value

e.g. (m = 200):

h(k) = k mod 200 if k is an integer;

h("NY") = (ASCII("N") x ASCII("Y")) mod 200

h("YN") = (ASCII("Y") x ASCII("N")) mod 200

Chained-hash-init(T): initialize the hash table T
Chained-hash-insert(T, x): insert new item pointed by x (to head of list)
Chained-hash-search(T, k): search for an element with key k in T[h(k)]
Chained-hash-delete(T, k): delete element with key k from the list T[h(k)]

```
struct node{
    element e;
    node * next;
 };

node * T[m];
```

```
Chained-hash-search(T, k) {
    node * p = T[h(k)];
    while ((p ≠ NULL) and (key(p→e) ≠ k))
        p = p→next;
    return p;
}
```

O(r) where r: length of list

```
Chained-hash-insert(T, x) {
    node * p = T[h(key(x →e))];
    Insert(p, x);     // Check duplicate?
}
```

O(1) or O(r) if need to check duplicate

```
Chained-hash-init(T) {
    for i = 0 to m-1 do
        T[i] = NULL;
}
```

O(m)

```
Chained-hash-delete(T, k) {
    search for x with key k;
    delete x;
}
```

O(r)

More detailed analysis on searching

m: size of hash table T;

n: no. of keys(elements) in T

What is r? Worst case: r = O(n)

e.g. K = {"amy", "apple", "avina", "alpha"..}

h(k) = ASCII(first letter) mod m

Well, it depends on hash function

Time complexity (worst case): (1) $\Theta(1)$ for computing hash value; (2) $\Theta(r)$ for traversing the list => $\Theta(1+r)$

Q: how about space?

What's a good hash function?

Ideal case (simple uniform hashing): Any given key is equally likely to hash into any of the m slots, independently of where any other key has hashed to.

=> expected $r \approx n/m$

Let $\alpha = n/m$ be called the load factor of T for storing n keys

Theorem: So, under the assumption of simple uniform hashing, it can be proved that average time complexity for searching is $\Theta(1+\alpha)$

=> If $\alpha=O(1)$ (i.e. $n = O(m)$), then (average) cost = $O(1)$

Hash function

Reminder: (1) Roughly speaking, a good hash function should distribute keys evenly into m slots, but not easy to find; (2) fast to compute

The division method

Is this assumption ok?

Assuming that keys are natural numbers

$h(k) = k \bmod m$

e.g. keys are strings
"pt" can be interpreted as a radix-128 integer:
$(112 \times 128)+116=14452$

Note the choice of m is important.

Example, take $m = 2^p$ where $p = 4$, is it a good choice?

No, it does not depend on all bits of the key!

Take m = prime not too close to power of 2

Example: if $n = 2000$, let $\alpha \approx 3$, that is, there are about 3 keys per list, what m should we pick?

$m \approx 2000/3 \approx 667$ and $2^9 = 512$, $2^{10}=1024$, pick $m=701$

The multiplication method

Pick a constant $0 < A < 1$

$h(k) = \lfloor m\,(\underbrace{kA - \lfloor kA \rfloor}_{\text{fractional part of } kA})\rfloor$

Choice of m is <u>not</u> that important. Usually take $m=2^p$, [p most sig. bits from fractional part]

Choice of A is important

a) Multiple k by A
b) Take the fractional part of kA
c) Multiple the result by m
d) Take the integral part of the result

e.g. m=8, A=0.3
h(5) = 4

e.g. If A = 0.5

$kA - \lfloor kA \rfloor$ can only be 0.5 or 0. So, if m = 8, what slots will not be used?

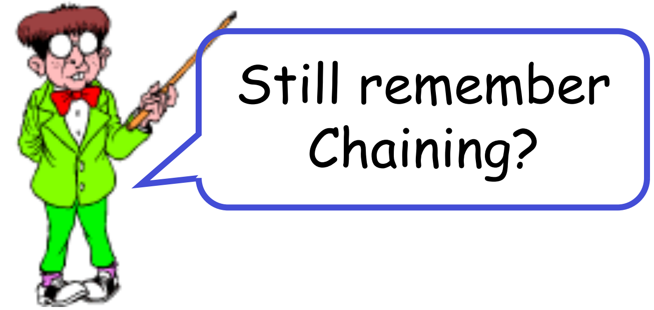
e.g. If A = 0.4

k ends with	$kA - \lfloor kA \rfloor$	h(k)
1	0.4	3
2	0.8	6
3	0.2	1
4	0.6	4
5	0.0	0
6	0.4	3
7	0.8	6
8	0.2	1
9	0.8	4
0	0.0	0

A should be close to an irrational number such as Golden ratio

$$A \approx \frac{\sqrt{5}-1}{2}$$

An alternative method for solving collisions: Open Addressing



All elements are stored in the hash table
i.e. T[i] = element x or NIL
No lists stored outside the table

No. of records stored in T (n) ≤ No. of slots in T (m)
load factor $\alpha \le 1$

Advantage of this scheme?

It avoids pointers and linked lists, so save space

How to handle collision?

key(x) uniquely determines this sequence

Idea: Given x, we can compute a sequence of slot numbers <s0, s1, s2, ...> Check if T[s0] is occupied. If no, put x there, otherwise, check T[s1], T[s2], ... until an empty slot is found or conclude that T is filled up already.

Probe sequence: can be computed from key(x)

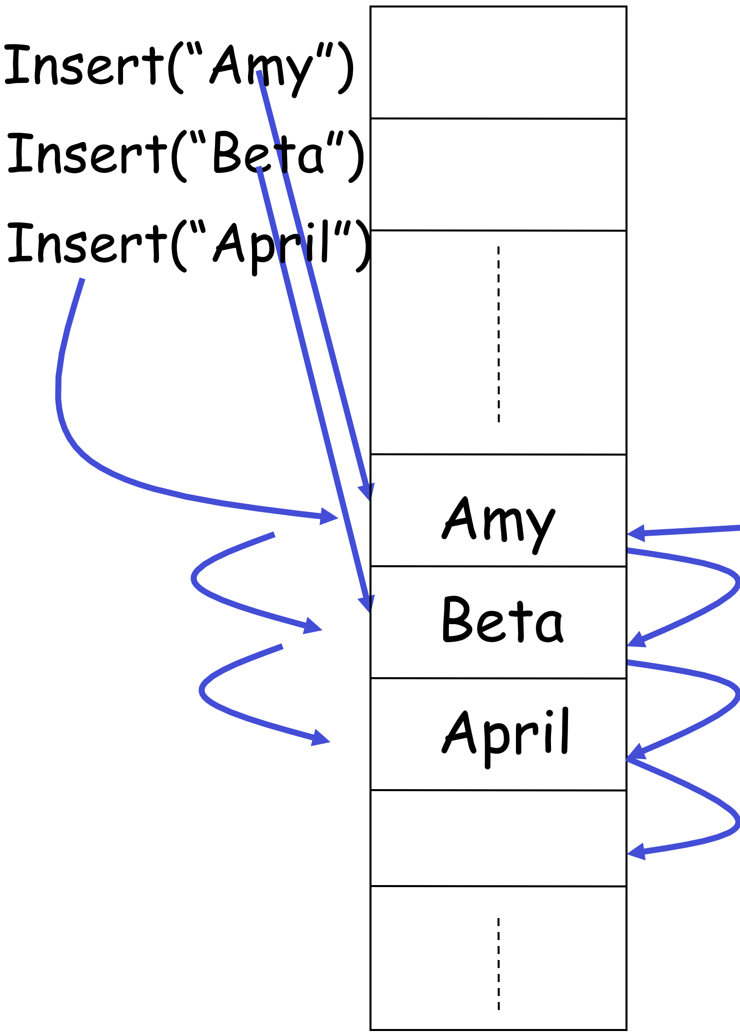
This procedure is called probing

Example:

Let h' be a hash function. Given x, we check if T[h'(key(x))] is occupied, then try T[h'(key(x))+1] mod m, T[h'(key(x))+2] mod m,... until an empty entry is found or T is full.

Probe sequence: <h'(key(x)), h'(key(x))+1 mod m, ...>

Example: k are names
h'(k) = ASCII(first letter of k) mod 26



Can you see how to do searching?

Follow the same probe sequence until the element is found or an empty slot is encountered

E.g. Search("April")

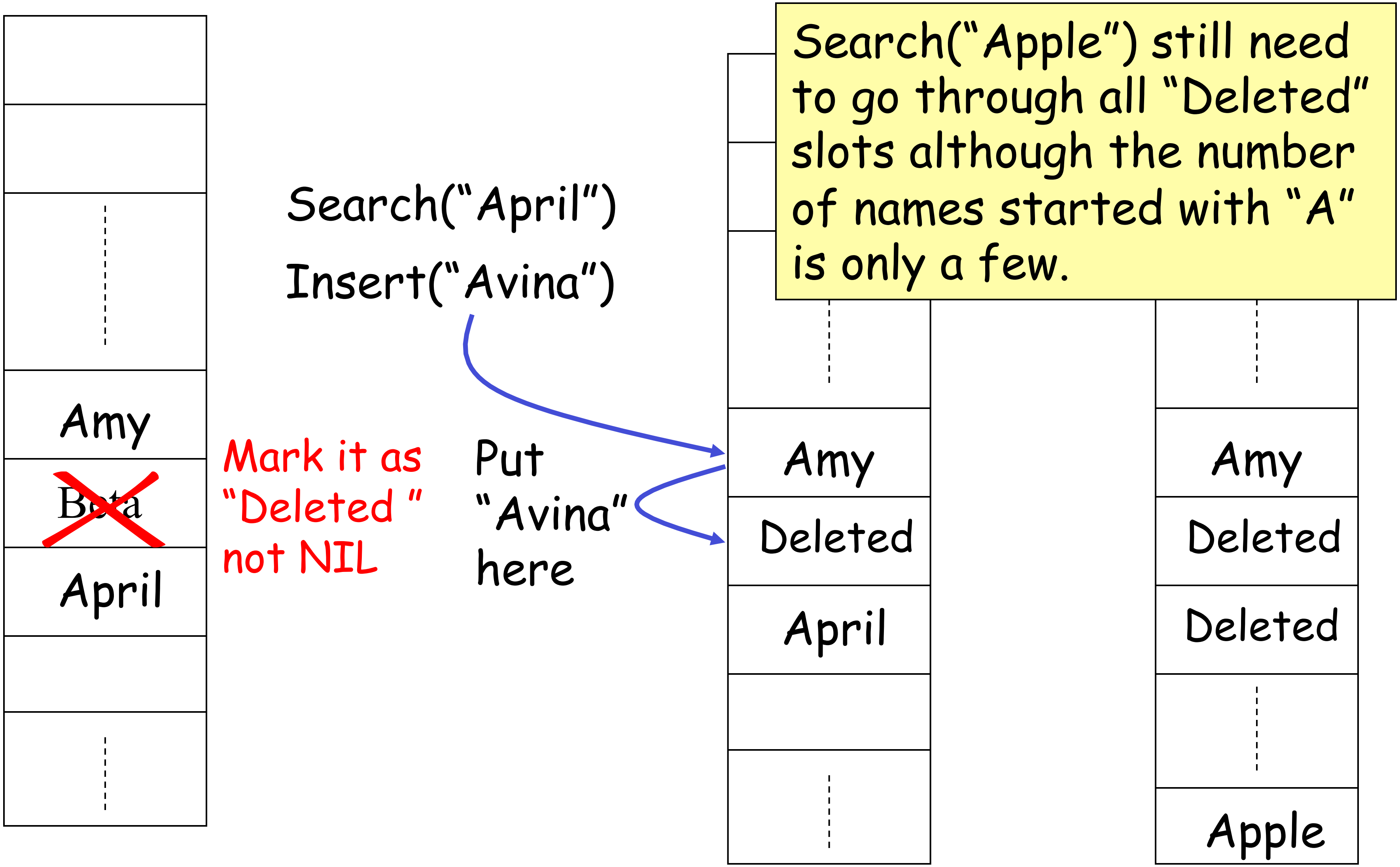
E.g. Search("Avina")

Is it ok?

Can you see how to do deletion?

One suggestion: Follow the same probe sequence until the element is found and mark it as NIL

e.g. Delete "Beta", then search "April"



May need to "reorganize" hash table if many deletions have occurred

We can extend hash function as follows:

Define $h(k, i)$ as $(i+1)$th entry in the probe sequence for key k

where $h(k, i): U \times \{0, 1, \ldots, m-1\} \rightarrow \{0, 1, \ldots, m-1\}$

The probe sequence is $\langle h(k, 0), h(k,1), \ldots, h(k, m-1) \rangle$

Our example probing (linear probing):

$h(k, i) = (h'(k) + i) \bmod m$

```
Hash-Insert(T, x) {
    i = 0
    repeat  j = h(key(x), i)
            if T[j] = Nil or Deleted
                T[j] = x; return
            else
                i = i + 1
    until i = m
    error "hash table overflow"
}
```

```
Hash-Search(T, k) {
    i = 0
    repeat  j = h(k, i)
            if T[j].key = k
                return j
            else
                i = i + 1
    until T[j] = Nil or i = m
    return Nil
}
```

```
Hash-Deleted(T, k) {
    i = 0
    repeat  j = h(k, i)
            if T[j].key = k
                Mark T[j] as "Deleted"; return
            else
                i = i + 1
    until T[j] = Nil or i = m
    return "Not found"
}
```

What is a good probe sequence?

Consider:
h(k, i) = (h′(k) + 2i) mod m
and for a particular k1, h′(k1) = 3, let m = 8

Although T has 4 free slots, the sequence generated by h(k1, i) does not check these free slots!

Nil
Occupied
Nil
Occupied
Nil
Occupied
Nil
Occupied

Requirement for h:
For every key k, the probe sequence <h(k, 0), h(k, 1), ..., h(k,m-1)> must be a permutation of <0, 1, ..., m-1> to make sure that every slot of T is checked

Review (1/3):

Major objective of "hashing" – to realize the "dictionary ADT" with O(1) [average case] search time

Idea:

Map the keys of records to [0..m-1] which represents indexes of an array.

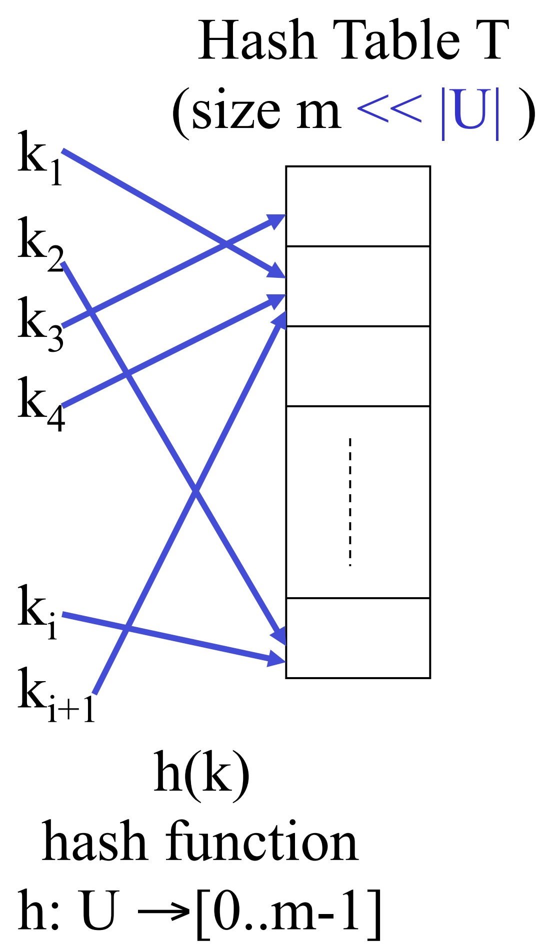
Remarks:

(i) 1-1 mapping will make the array too large (not practice or even infeasible)

(ii) To make space smaller, allow different keys map to the same array entry.

Collision unavoidable

Two issues:
- (a) Design a "good" hashing function
- (b) How to handle collisions



Review (2/3):

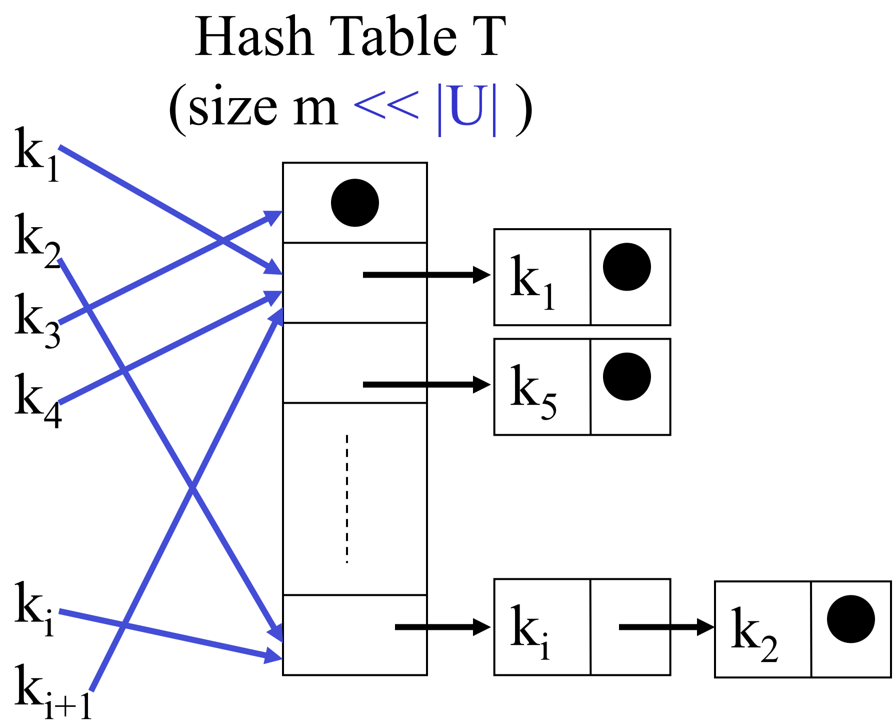
What is a “good” hash function?

Trying to satisfy the “simple uniform distribution”, i.e., each key has same probability being hashed to any of the array slot.

Two design methods
- (a) Division method (more common)
- (b) Multiplication method

How to handle collision?

Two approaches
- (a) Chaining
- (b) Open addressing



Review (3/3):

Open addressing:

Idea – put all records in the array as long as there are empty slots

Probe sequence (given a key k, this is fixed):

$\langle h(k, 0), h(k, 1), h(k, 2), \ldots, h(k, m-1)\rangle$ // array: [0..m-1]

General requirements for probe sequence:

MUST be a permutation of $\langle 0, 1, \ldots, m-1\rangle$ Otherwise (?)

E.g. $h(k,i) = (h'(k) + i) \bmod m$

where $h'(k)$ is any hash function // linear probing

First slot to try: $h'(k)$

Second slot to try: $h'(k) + 1$

Third slot to try: $h'(k) + 2$

....

E.g. $h(k, i) = (h'(k) + 3i) \bmod m$

[Does not satisfy the requirement: $m = 9$, $h'(k) = 2$ for some key k]

Computing probe sequence, h(k, i)

1 Linear probing

$h(k, i) = (h'(k) + i) \bmod m$
where $h'(k)$ is called an auxiliary hash function

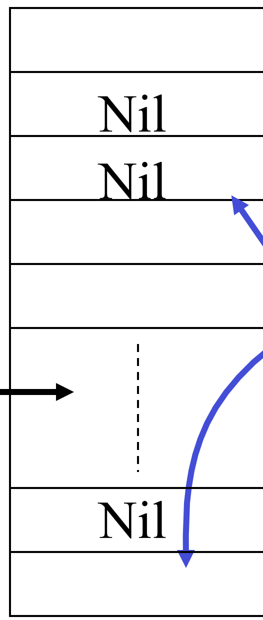
How many distinct sequences that can be generated by $h(k, i)$?

Ans: At most m

Problem with linear probing

Primary clustering: long runs of occupied slots build up

Any key hashed into the cluster will go through the cluster and add to the cluster



j occupied slots

Consider probability that this slot will be filled next

Prob = 1/m

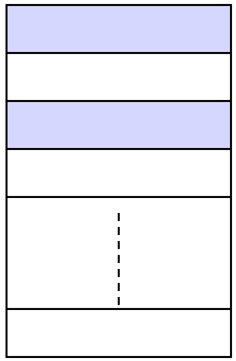
Assumption (Uniform hashing):
Each key is equally likely to hash into any of the m slots.

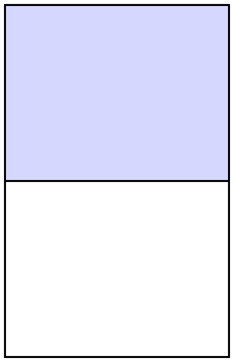
Prob = (j+1)/m

Implication:
Bigger cluster has bigger chance to grow and become even bigger faster

Note: Primary clustering makes insertion and searching inefficient

Example: Consider unsuccessful search with m/2 slots are occupied





Ave. # of probs
= (1/2)2+(1/2)1
= 1.5

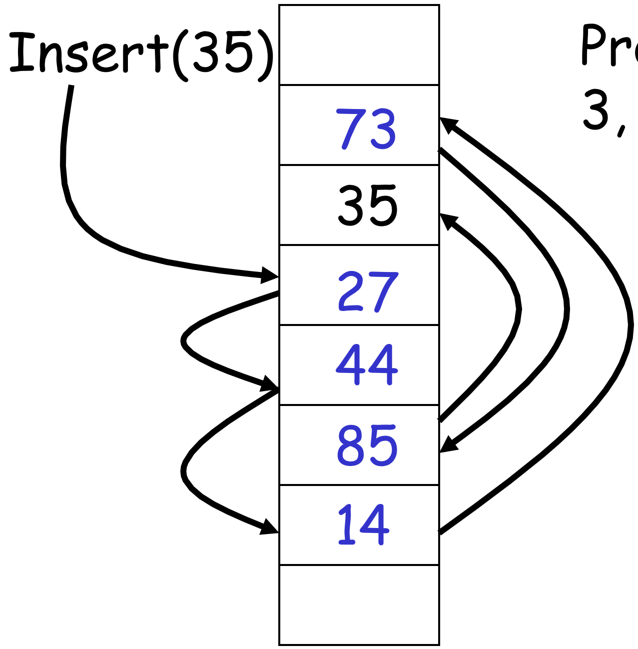
Ave. # of probs
= (1/2)(m/4+1)+(1/2)1
≈ m/8

Probe sequence generated by linear probing is not good, so.. to avoid "primary clustering", we can =>

2 Quadratic probing

$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$
where $h'(k)$ is an auxiliary hash function
c_1 and c_2 ($\neq 0$) are constants

e.g. if $h'(k) = k \bmod 8$, and $h(k, i) = (h'(k) + \frac{1}{2}(i + i^2)) \bmod 8$



Probe sequence:
3, 4, 6, 1, 5, 2, 0, 7

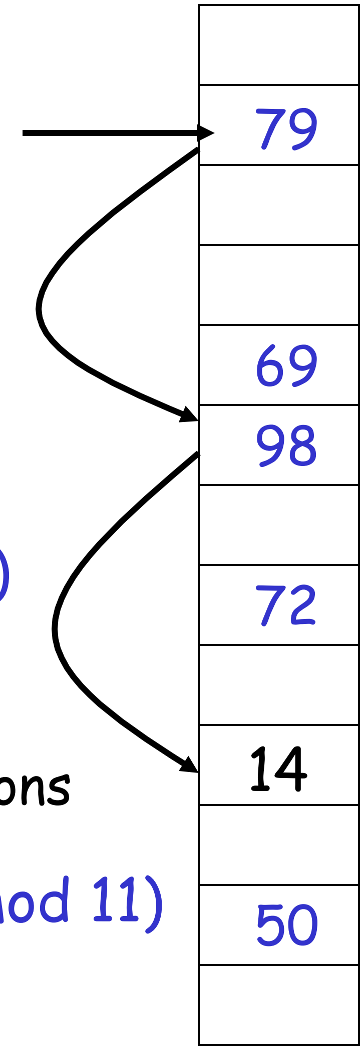
<u>Problem with quadratic probing</u>
Secondary clustering: If two keys have the same initial probe position, their probe sequences are the same

Again, quadratic probing uses only m distinct probe sequences

3 Double hashing: one of the best methods for open addressing

$h(k, i) = (h_1(k) + ih_2(k)) \bmod m$
where h_1 and h_2 are auxiliary hash functions

The probe sequence:
$\langle h_1(k), (h_1(k)+h_2(k)) \bmod m, \ldots \rangle$



> Note: in both linear and quadratic probing, the initial probe position determines the probe sequence!

How many probe sequences are used?

$\Theta(m^2)$

$h_1(k)$ determines initial probe position,
$h_2(k)$ determines offset for successive probe positions

Example: $m=13$, $h_1(k)=k \bmod 13$ and $h_2(k)=1+(k \bmod 11)$ and we want to insert(14)

<u>Remark</u>: $h_2(k) \neq 0$, and must be relatively prime to m for the entire table to be searched

Do you how to prove it?

Remark : With open addressing, when the table is getting full, we should build a larger table and rehash the elements there.