Hashing & Hash Tables

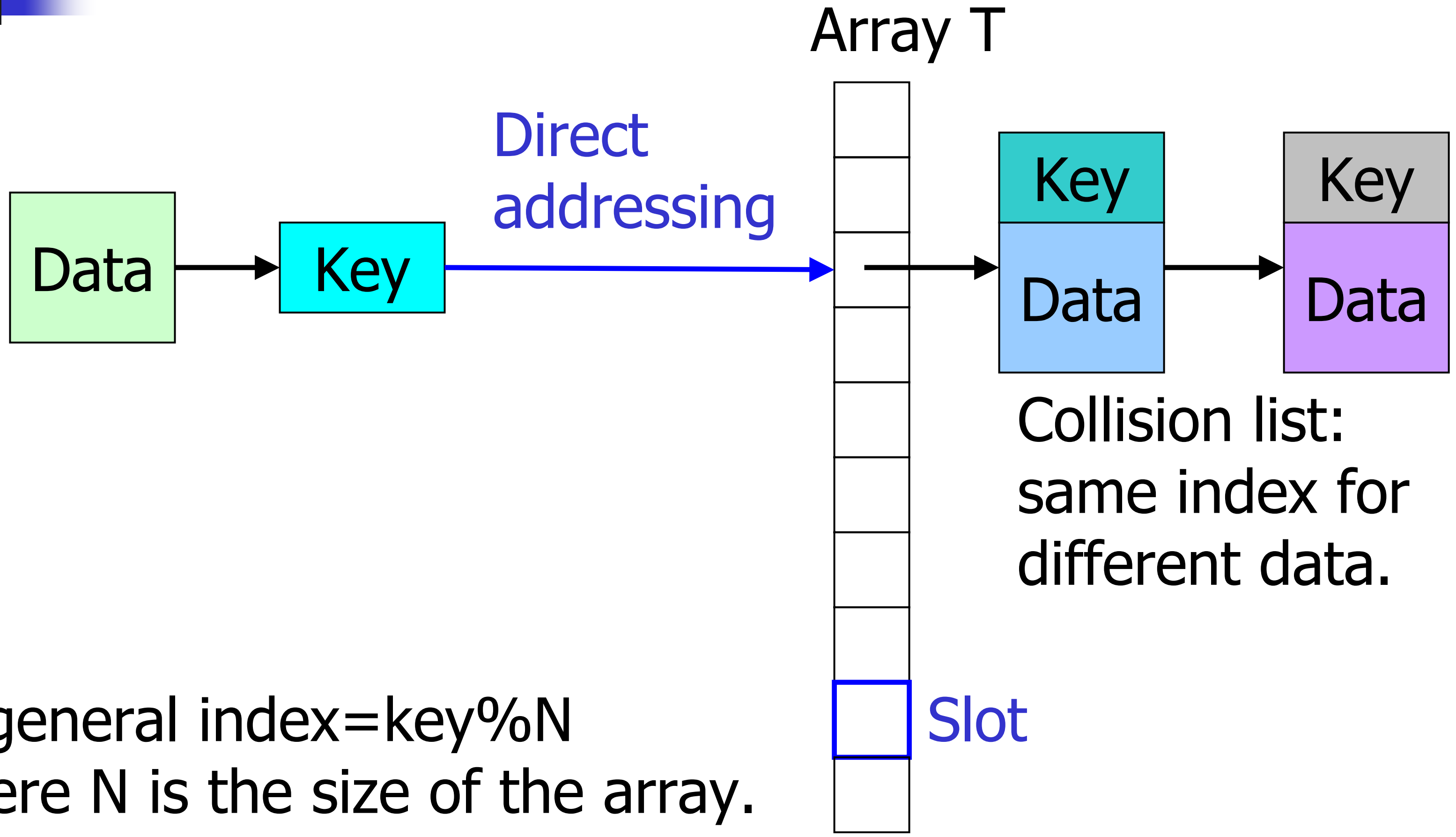
Alexandre David

B2-206

Introduction

- A *hash table* is an effective data structure for implementing *dictionaries* (set with insert, search, and delete operations).
- Worst case access time is *O(n)* but expected time is *O(1)*.
- Idea:
  - use direct addressing of arrays
  - compute an index from a key (i.e. hash value)
  - handle collisions with lists.

Hash Tables



Collision list: same index for different data.

In general index=key%N
where N is the size of the array.

Direct Access Tables

- Idea:
  - Suppose that the set of keys is $K \subseteq \{0,1,\ldots,m-1\}$, and keys are distinct.
  - Setup an array T[0…m-1]:
    T[k]=x if k∈K and key[x]==k
    T[k]=NIL otherwise.

$\Theta(1)$ time

Direct-Address Tables

- Work well for a small set of (different) keys.
  - Direct-address table (i.e. array) where each *slot* corresponds to a key.
  - Problem with the range of the key.

```
search(T,k):
return T[k]
```

```
insert(T,x):
T[key(x)]=x
```

```
delete(T,x):
T[key(x)]=NIL
```

Hash Tables

- How to store if the set of keys is large?
  - Use a hash function to map keys to slots: *collisions* solved by *chaining*.

```
search(T,k):
return List_search(T[h(key(x))])
```

```
insert(T,x):
List_insert(T[h(key(x))],x)
```

```
delete(T,x):
List_delete(T[h(key(x))],x)
```

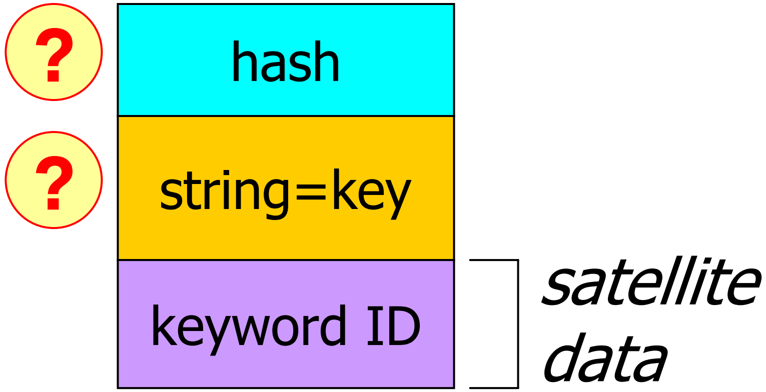
Application: Symbol-Table

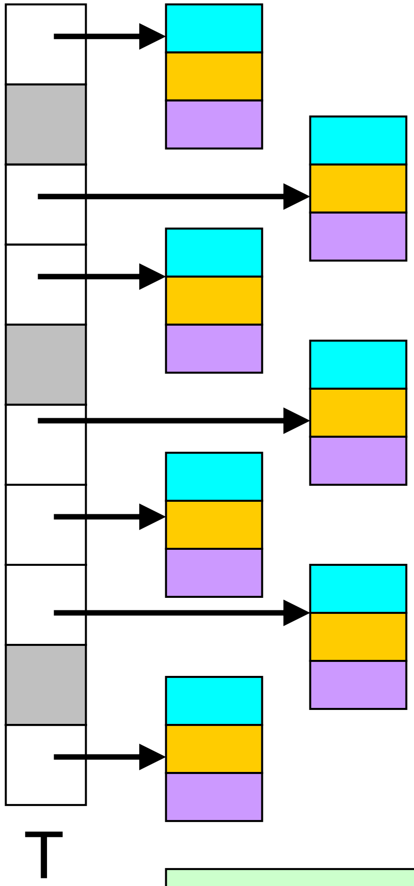
In any reasonable lexical analyzer.

Input: a string.
Output: is it a keyword
and if yes which one?

Symbol table **T** holds *n* **records**.
Direct address table.

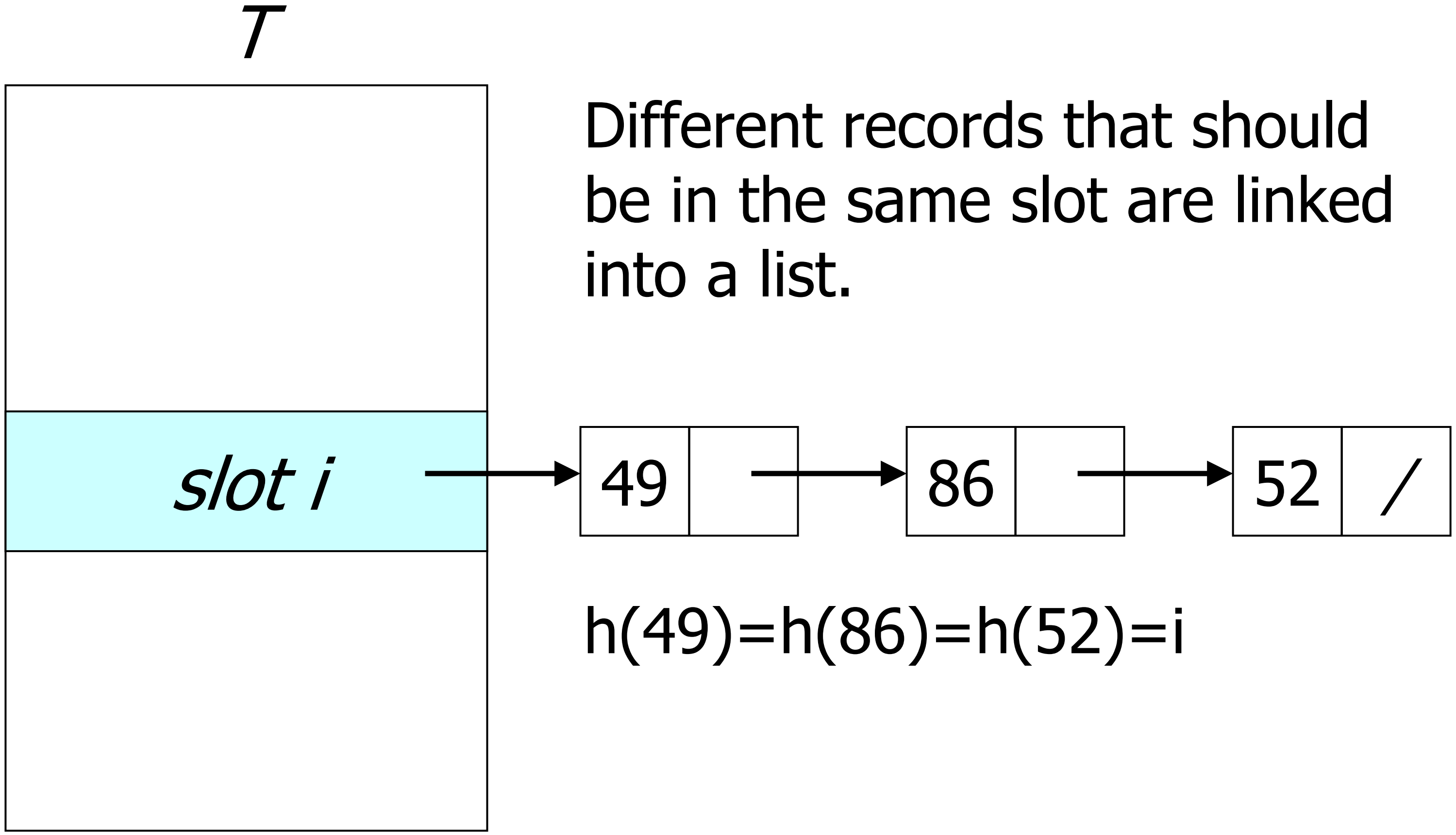
Record:





See gperf

Resolving Collision by Chaining



Different records that should be in the same slot are linked into a list.

h(49)=h(86)=h(52)=i

Analysis of Chaining

- Assume *simple uniform hashing:*
  - Each key is equally likely to be hashed to any slot of table T, independently of where other keys are hashed.
- Let n be the number of keys in the table and m the number of slots.
- Define the load factor of T to be $\alpha = n/m$.
  - Represents the average number of keys per slot.

Search Cost

- Expected time to search for a record with a given key=$\Theta(1+\alpha)$.

Apply hash
function and
access slot.

Search the
list.

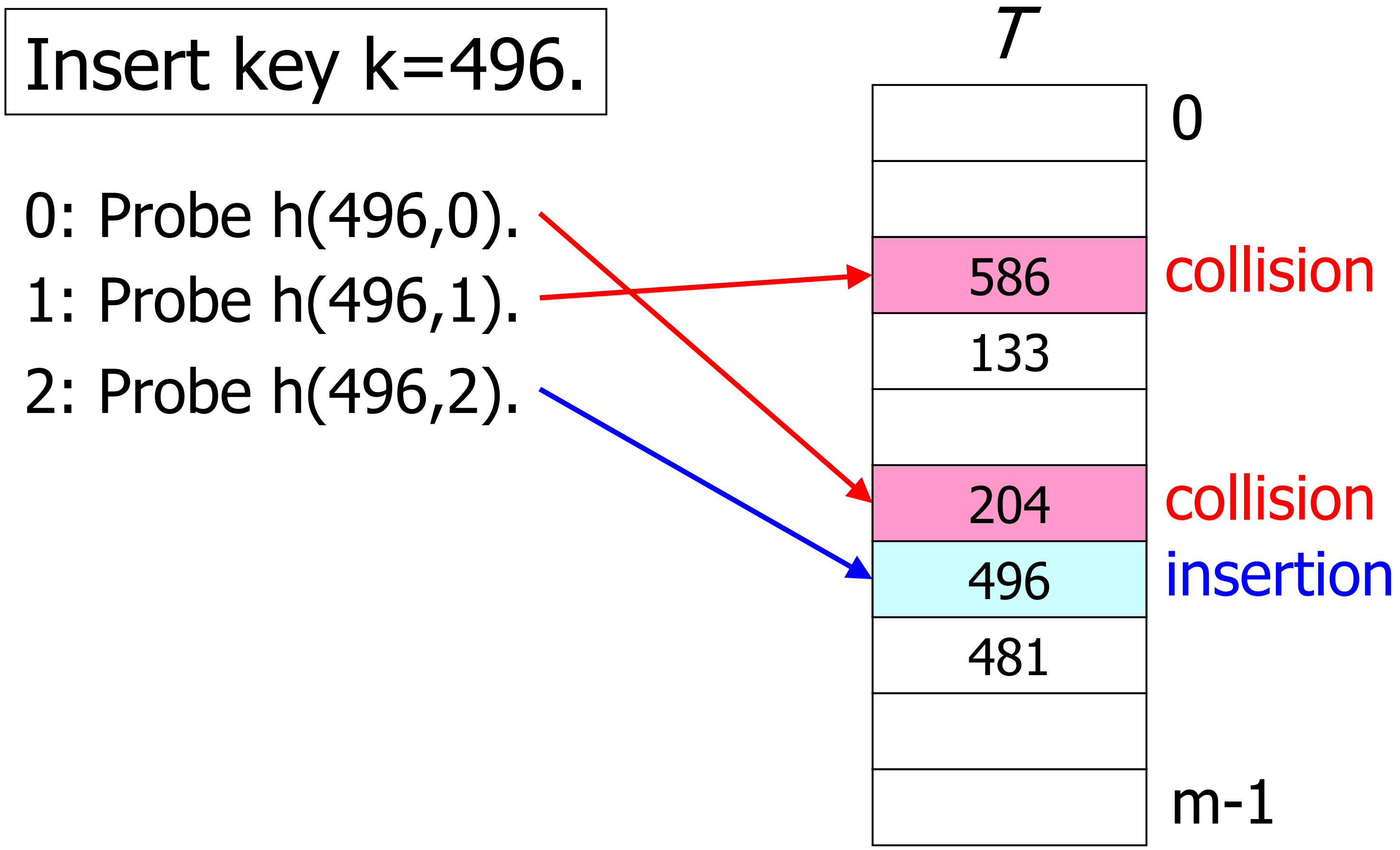
- Expected time=$\Theta(1)$ if $\alpha=O(1)$, or equivalently if $n=O(m)$.
  - We can enforce this by *re-hashing*.

Resolving Collisions by Open Addressing

- Idea: No storage is used outside of the hash table itself.
  - Insertion probes the table until an empty slot is found.
  - The hash function depends on the key and the probe number.
    $h: U \times \{0,1,..,m-1\} \rightarrow \{0,1,..,m-1\}$.
  - The probe sequence $\langle h(k,0),h(k,1),..,h(k,m-1)\rangle$ is a permutation of $\{0,1,..,m-1\}$.
  - Problem: The table may fill up and deletion is difficult.

Open Addressing

Insert key k=496.



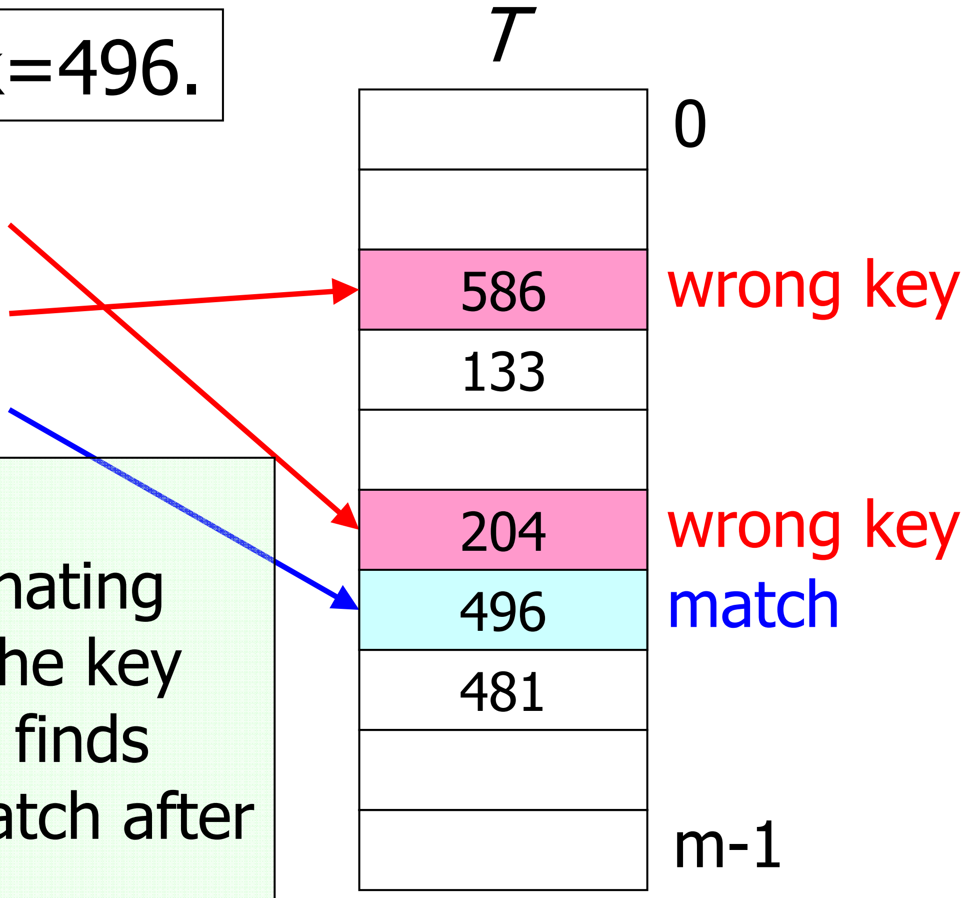
0: Probe h(496,0).

1: Probe h(496,1).

2: Probe h(496,2).

Open Addressing

Search for key k=496.



0: Probe h(496,0).

1: Probe h(496,1).

2: Probe h(496,2).

Search uses the same probe sequence, terminating successfully if it finds the key and unsuccessfully if it finds an empty slot or no match after m tries.

Open Addressing

```
Hash_insert(T,k):
i = 0
repeat
    j = h(k,i)
    if T[j] == NIL then
        T[j] = k
        return j
    fi
    i = i+1
until i == m
error
```

```
Hash_search(T,k):
i = 0
repeat
    j = h(k,i)
    if T[j] == k then
        return j
    fi
    i = i+1
until T[j] == NIL or i == m
return NIL
```

Probing Strategies

- Linear probing:
  - Given an ordinary hash function $h'(k)$, linear probing uses the hash function $h(k,i)=(h'(k)+i) \bmod m$.
  - Simple method.
  - Suffers from primary clustering, where long runs of occupied slots build up, increasing the search time. Moreover, these long runs tend to get longer!

Probing Strategies

- Double hashing: (as in example)
  - Given two ordinary hash functions $h_1(k)$ and $h_2(k)$, double hashing uses the hash function $h(k,i)=(h_1(k)+i*h_2(k)) \bmod m$.
  - Generally produces excellent results, but $h_2(k)$ must be relatively prime to m. One way: Make m a power of 2 and design $h_2(k)$ to produce only odd numbers.

Analysis of Open Addressing

- Assume *uniform hashing:*
  - Each key is equally likely to have any one of the $m!$ permutations as its probe sequence.
  - Theorem:
    Given an open-addressed hash table with load factor $\alpha=n/m<1$, the expected number of probes in an unsuccessful search is at most $1/(1-\alpha)$.
  - Note: We can use re-hashing to maintain $\alpha<1$.

Analysis of Open Addressing

- Implications of the theorem:
  - If α is constant then accessing an open-addressed hash table takes constant time.
  - If the table is half full then the expected number of probes is 1/(1-0.5)=2.
  - If the table is 90% full then the expected number of probes is 1/(1-0.9)=10.

Hash Functions

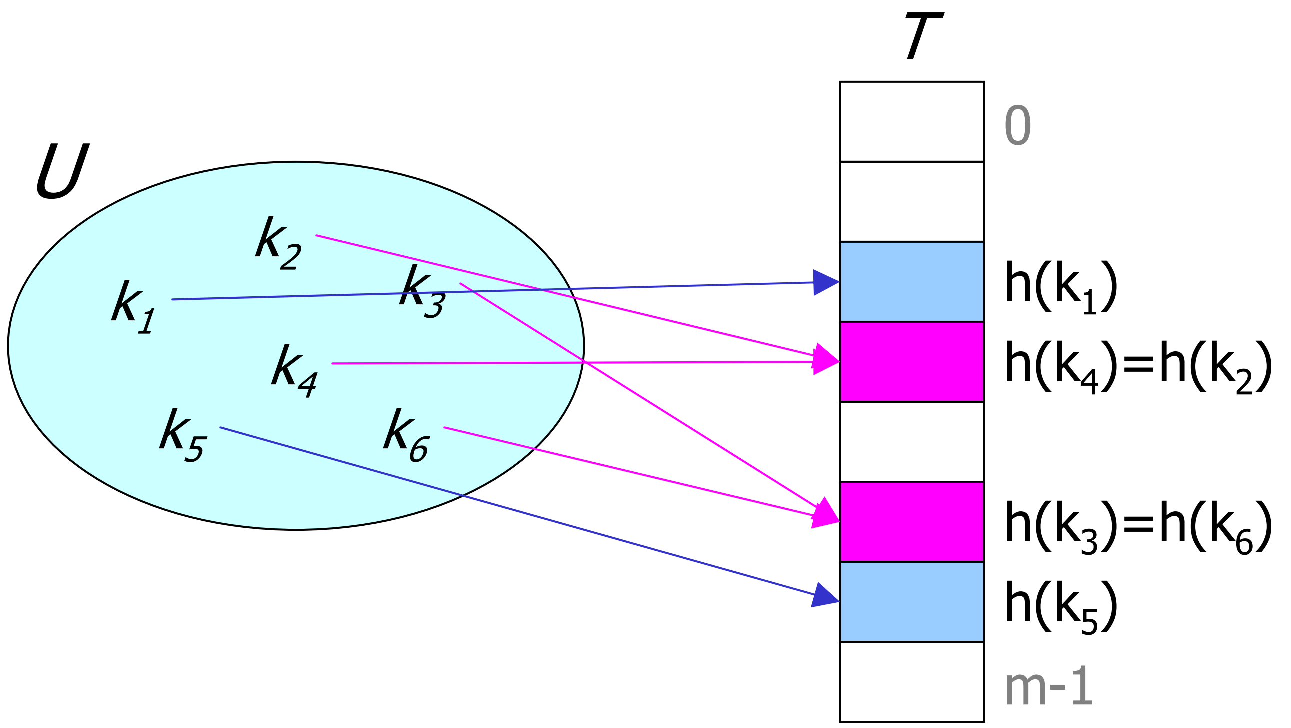
- What makes a good hash function?
- If we know the keys in advance then it is possible to construct a perfect hash function and hash table.
  - We cheat when we can.

?

- But what if we don’t know the keys or even the number of elements to be stored?

Hash Functions

Solution: Use a hash function *h* to map the universe *U* of all keys into *{0,1,…,m-1}*:



When a record to be inserted maps to an occupied slot, a collision occurs.

Choosing a Hash Function

- Hard to guarantee the assumption of simple uniform hashing! Several common techniques work well in practice as long as their *deficiencies* can be avoided.
- Want we want:
  - A good hash function should distribute the keys uniformly into the slots of the table.
  - Regularity in the key distribution should not affect this uniformity.

Division Method

- Assume all keys are integers and define $h(k)=k \bmod m$.
- Deficiency: Don't pick an m that has a small divisor d. Keys that are congruent modulo d can affect uniformity. Typically, choose m prime.
- Extreme deficiency: If $m=2^r$ then the hash doesn't even depend on all the bits of k !

Division Method

- Pick m to be a prime not too close to a power of 2 or 10 and not otherwise used prominently in your computing environment.
- The catch: It may be inconvenient to make the table size a prime.
    - Popular method in practice.

Multiplication Method

- Assume that all keys are integers, $m=2^r$, and our computer has *w-bit* words. Define $h(k)=(A*k \bmod 2^w) >> (w-r)$, where A is an odd integer $2^{w-1}<A<2^w$.
- Don't pick A too close to 2^w.
- Fast operations.
- Effect: Mix the bits.

Dot-product Method

- Take a randomized strategy.
  - Let *m* be prime. Decompose key *k* into *r+1* digits, each with value in the set {0,1,…,m-1}: $k=\langle k_0,k_1,\ldots,k_r\rangle$ with $0 \le k_i < m$ – *k in base m*.
  - Pick $a=\langle a_0,a_1,\ldots,a_r\rangle$ where a_i is chosen randomly from {0,1,…,m-1} – *a random in base m*.
  - Define $$h_a(k)=\sum_{i=0}^{r} a_i k_i \bmod m$$
  - Excellent in practice by expensive to compute.

2 vectors in base m

dot-product

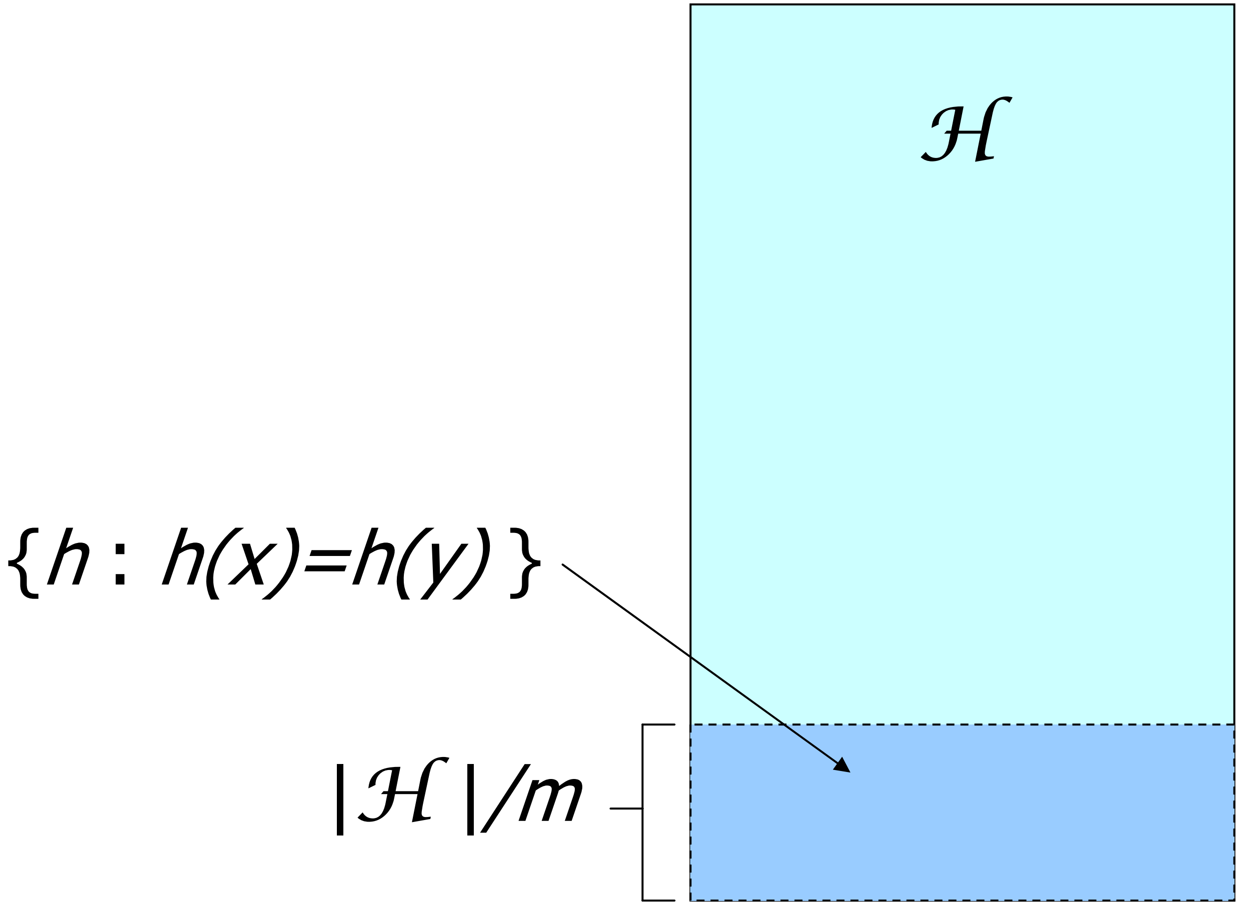
Weakness of Hashing

- For any hash function h, a set of keys exists that can cause the average access time to skyrocket (linear).
  - An adversary can pick all keys from $\{k \in U : h(k)=i\}$ for some slot *i*.
- **Idea**: Choose the hash function at random, independently from the keys!
  - Even if an adversary sees the code, she cannot find bad keys since she doesn't know which hash function will be used.

Universal Hashing

- Definition: Let U be a universe of keys and $\mathcal{H}$ be a finite collection of hash functions (mappings $U \rightarrow \{0,1,\ldots,m-1\}$). $\mathcal{H}$ is universal if for all $x,y \in U$ where $x \neq y$, we have $|\{h \in \mathcal{H} : h(x)=h(y)\}| = |\mathcal{H}|/m$.
  - The chance of a collision between x and y is $1/m$ if we choose h randomly from $\mathcal{H}$.

Universal Hashing



$\mathcal{H}$

$\{h : h(x)=h(y)\}$

$|\mathcal{H}|/m$

Universality is Good™

- Theorem:
  Let h be a hash function chosen (uniformly) at random from a universal set $\mathcal{H}$ of hash functions.
  Suppose h is used to hash n arbitrary keys into the m slots of a table T.
  Then, for a given key x, we have
  E[#collisions with x] $< n/m$.

Proof

- Let C_x be the random variable denoting the total number of collisions of keys in T with x. C_x counts collisions with x.
- Let $c_{xy}=1$ if $h(x)=h(y)$, 0 otherwise. Indicator variable.
- Notes:
  $E[c_{xy}]=1/m$
  $$C_x = \sum_{y \in T-\{x\}} c_{xy}$$

Proof (cont.)

$$E[C_x] = E\left[\sum_{y \in T - \{x\}} c_{xy}\right]$$

$$= \sum_{y \in T - \{x\}} E[c_{xy}]$$

$$= \sum_{y \in T - \{x\}} 1/m = \frac{n-1}{m} < \frac{n}{m}$$

How to Construct a Set of Universal Hash Functions?

- Randomized strategy:
  - Let *m be prime*. Decompose key *k* into *r+1* digits, each with value in the set {0,1,…,m-1}: $k=\langle k_0,k_1,\dots,k_r\rangle$ with $0\le k_i<m$ – *k in base m*.
  - Pick $a=\langle a_0,a_1,\dots,a_r\rangle$ where a_i is chosen randomly from {0,1,…,m-1} – *a random in base m*.
  - Define $$h_a(k)=\left(\sum_{i=0}^{r}a_ik_i\right)\bmod m$$
  - How big is $\mathcal{H}=\{h_a\}$? $|\mathcal{H}|=m^{r+1}$.

Dot-product modulo m.

Dot-product Hash Functions Are Universal!

- **Theorem**: The set $\mathcal{H}=\{h_a\}$ is universal.
- Proof:
  - Suppose $x=\langle x_0,x_1,\ldots,x_r\rangle$ and $y=\langle y_0,y_1,\ldots,y_r\rangle$ be distinct keys. They differ in at least one digit.
  - For how many $h_a\in\mathcal{H}$ do x and y collide?

    $h_a(x)=h_a(y)$ implies

$$\sum_{i=0}^{r} a_i x_i \equiv \sum_{i=0}^{r} a_i y_i \pmod{m}$$

Proof (cont.)

$$\sum_{i=0}^{r} a_i x_i \equiv \sum_{i=0}^{r} a_i y_i \pmod m$$

- For every choice of r a_i, only one value of the last a_j will cause the collision. Number of h_a causing the collision is $m^r = |\mathcal{H}|/m$.

In Practice

- If you know almost nothing on the elements to be stored (size, number…),
  - you need for a fast good hash function, maybe several ones,
  - you need dynamic hash tables,
  - it's convenient to have the size being a power of 2,
  - and you should check http://burtleburtle.net/bob/hash/

Code Example - Search

Size=2^p

```
typedef unsigned int uint;

typedef struct elem_s {
    struct elem_s *next;
    uint hashValue;
    data_t key;
} elem_t;

typedef struct {
    elem_t **slots;
    uint mask;
    uint n;
} table_t;
```

```
const elem_t* search(const table_t* t,
                                    const data_t* k) {
    uint h = hash(k);
    const elem_t *e;
    for(e = t->slots[h & t->mask];
          e != NULL &&
             !(e->hashValue == h &&
                strcmp(k, &e->key,
                           sizeof(data_t)) == 0);
          e = e->next);
    return e;
}
```

Rehash

Size=2^p

```
typedef unsigned int uint;

typedef struct elem_s {
    struct elem_s *next;
    uint hashValue;
    data_t key;
} elem_t;

typedef struct {
    elem_t **slots;
    uint mask;
    uint n;
} table_t;
```

```
void rehash(table_t *t) {
    uint old_size = t->mask+1;
    uint i, new_size = old_size << 1;
    uint new_mask = new_size – 1;
    elem_t **slots = (elem_t**)
            calloc(new_size, sizeof(elem_t*));
    for(i = 0; i < old_size; ++i) {
        elem_t *e = t->slots[i];
        while(e != NULL) {
            elem_t *next = e->next;
            uint j = e->hashValue & new_mask;
            e->next = slots[j];
            slots[j] = e;
            e = next;
        }
    }
    free(t->slots);
    t->slots = slots;
    t->mask = new_mask;
}
```