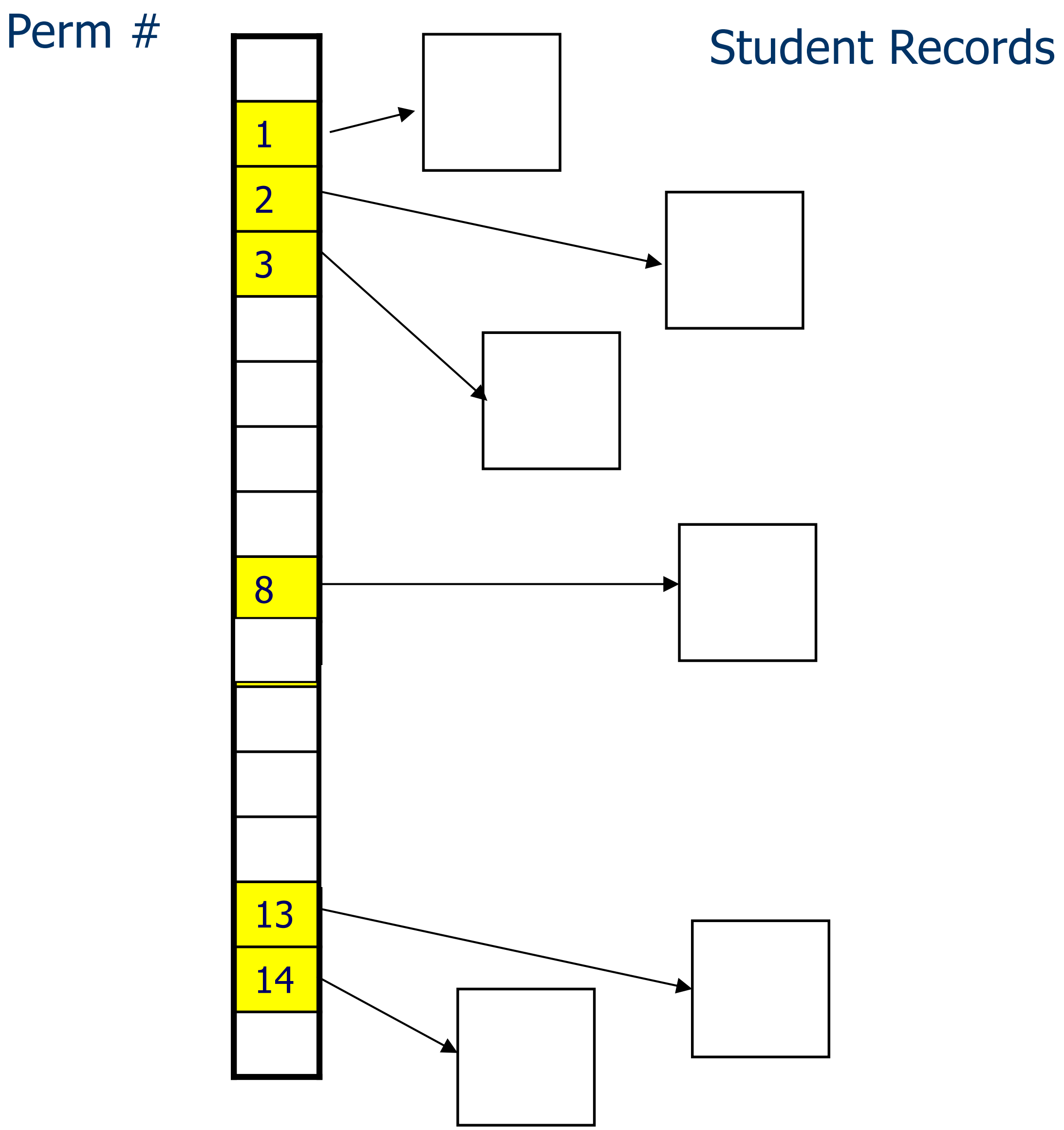
Hash Tables: Intuition

- Hashing is function that maps each key to a location in memory.

- A key's location does not depend on other elements, and does not change after insertion.
    - unlike a sorted list
- A good hash function should be easy to compute.

- With such a hash function, the dictionary operations can be implemented in O(1) time.

One Simple Idea: Direct Mapping



Hashing : the basic idea

- Map key values to hash table addresses

  keys -> hash table address

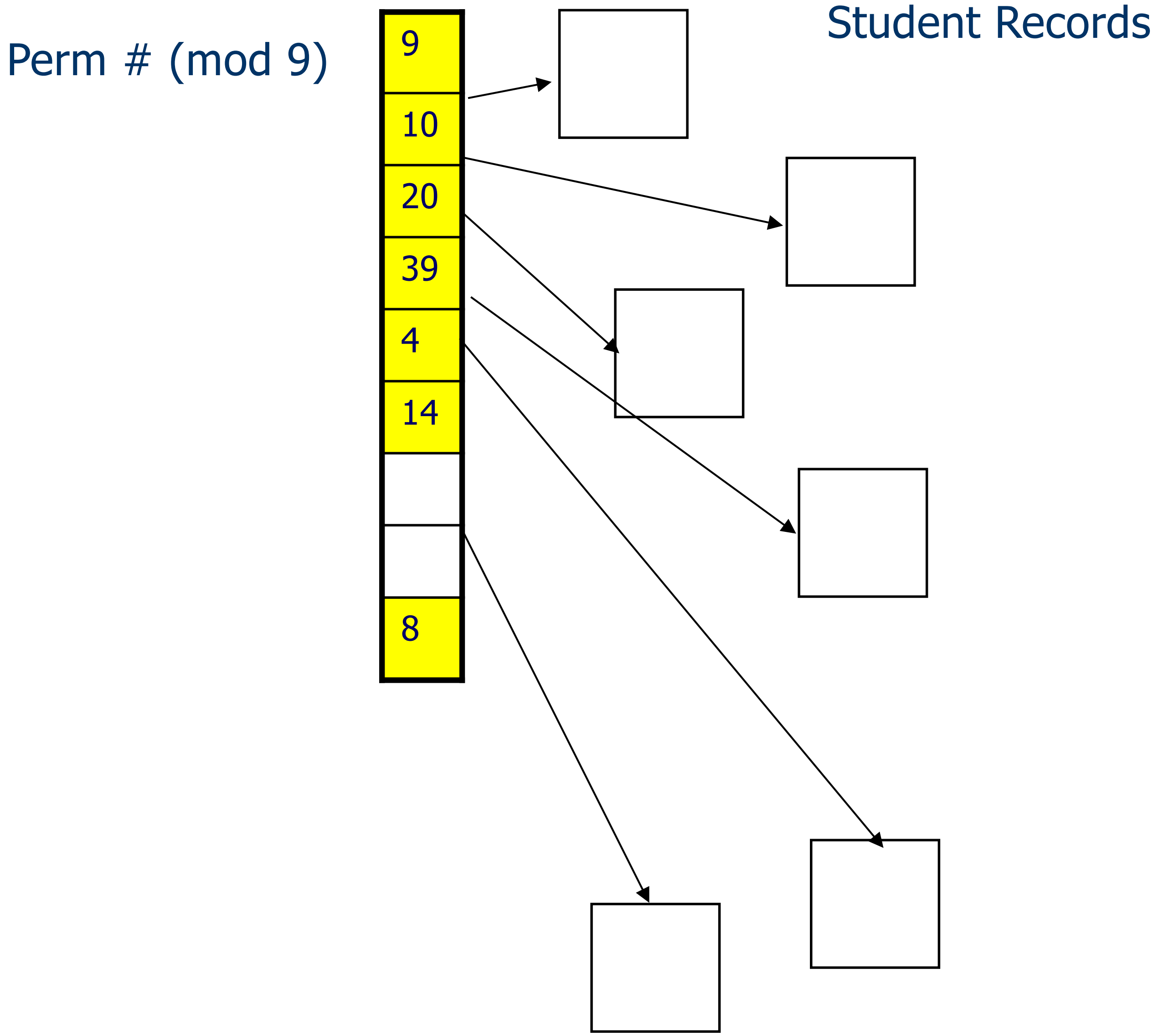
  This applies to find, insert, and remove

- Usually: *integers* -> {0, 1, 2, …, *Hsize*-1}

  Typical example: $f(n) = n \bmod Hsize$

- Non-numeric keys converted to numbers
  - □ *For example, strings converted to numbers as*
    - Sum of ASCII values
    - First three characters

Hashing : the basic idea



Hashing:

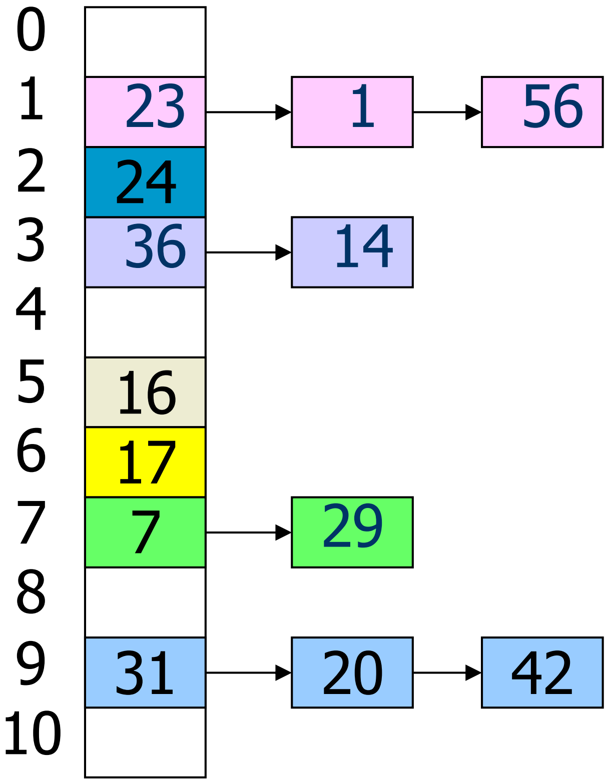
- *Choose a hash function h; it also determines the hash table size.*
- *Given an item x with key k, put x at location h(k).*
- *To find if x is in the set, check location h(k).*
- *What to do if more than one keys hash to the same value. This is called collision.*
- *We will discuss two methods to handle collision:*
  - Separate chaining
  - Open addressing

Separate chaining

- Maintain a list of all elements that hash to the same value
- Search -- using the hash function to determine which list to traverse

```
find(k,e)
   HashVal = Hash(k,Hsize);
   if (TheList[HashVal].Search(k,e))
   then return true;
   else return false;
```

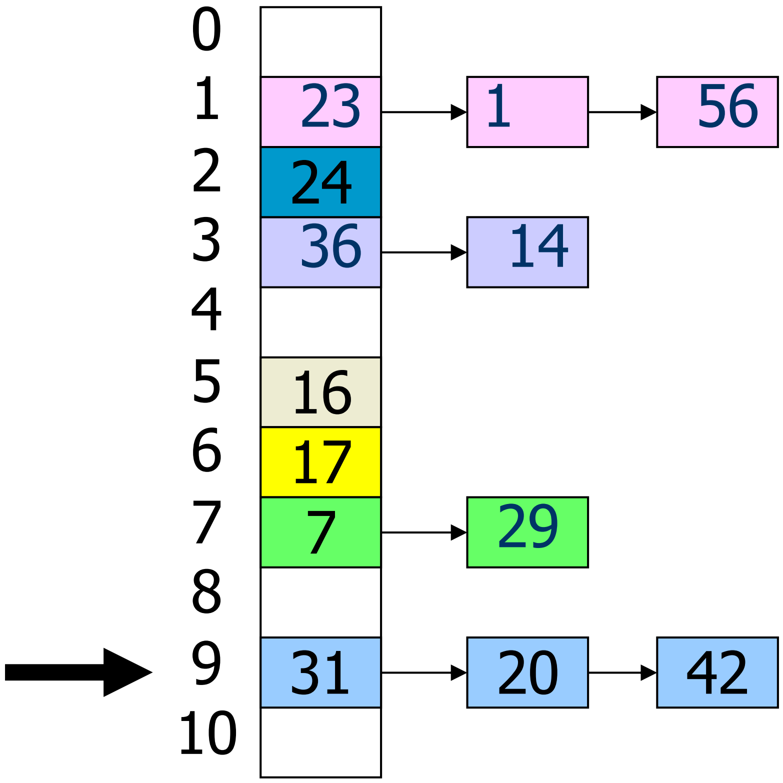
- Insert/deletion–once the “bucket” is found through *Hash*, insert and delete are list operations

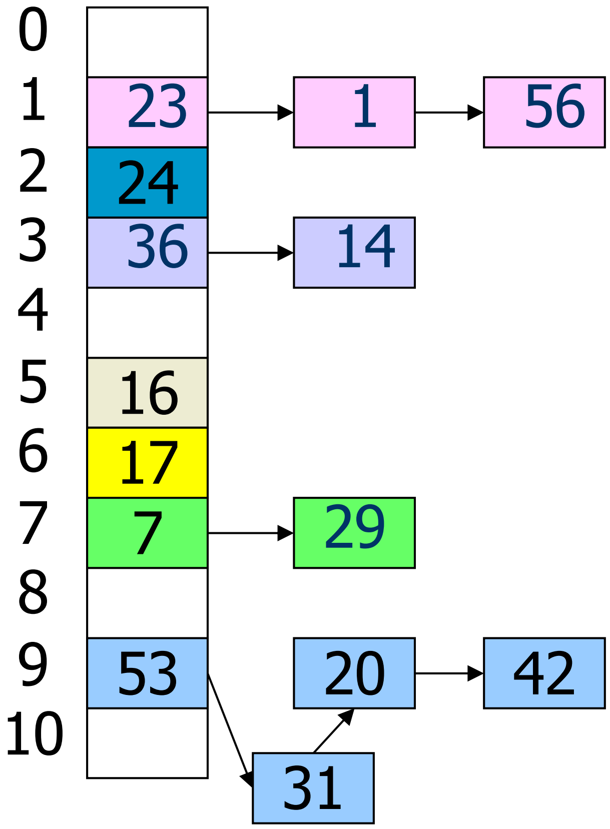


```
class HashTable {
  ……
  private:
    unsigned int Hsize;
    List<E,K> *TheList;
    ……
```

Insertion: insert 53

53 = 4 x 11 + 9
53 mod 11 = 9





Analysis of Hashing with Chaining

- Worst case
  - All keys hash into the same bucket
  - a single linked list.
  - insert, delete, find take O(n) time.
- Average case
  - Keys are uniformly distributed into buckets
  - O(1+N/B): N is the number of elements in a hash table, B is the number of buckets.
  - If N = O(B), then O(1) time per operation.
  - N/B is called the load factor of the hash table.

Open addressing

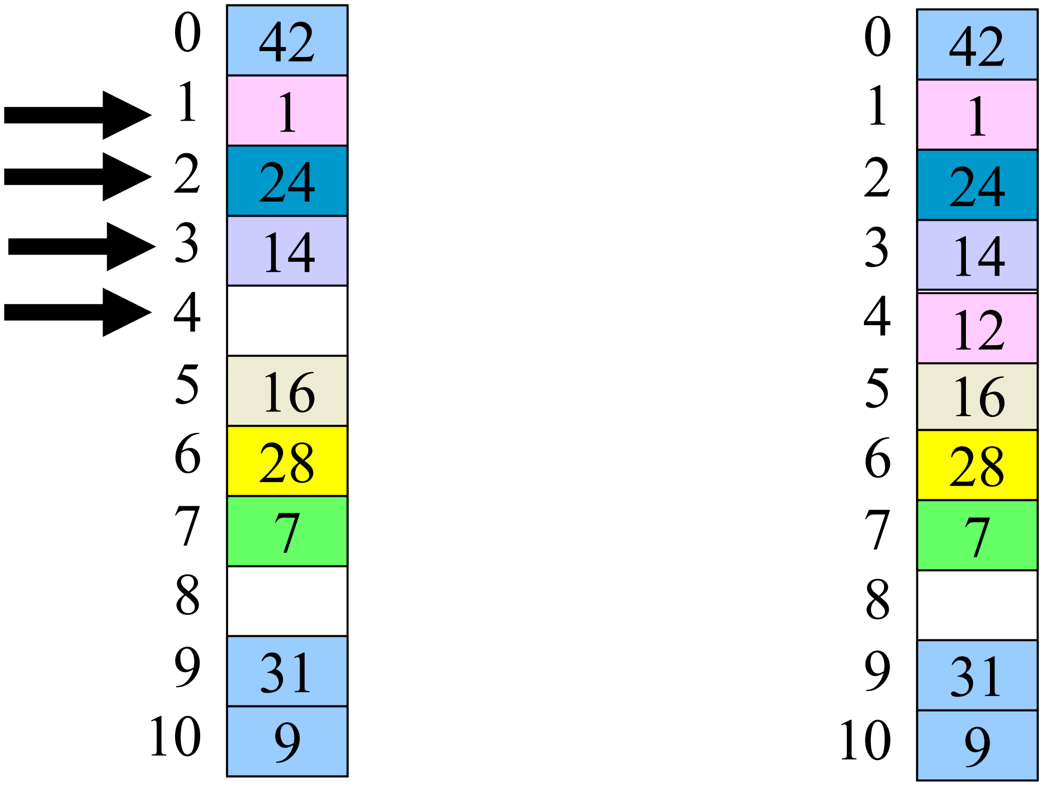
- If collision happens, alternative cells are tried until an empty cell is found.

- Linear probing :
  Try next available position

0	42
1	1
2	24
3	14
4	
5	16
6	28
7	7
8	
9	31
10	9

Linear Probing (insert 12)

12 = 1 x 11 + 1
12 mod 11 = 1

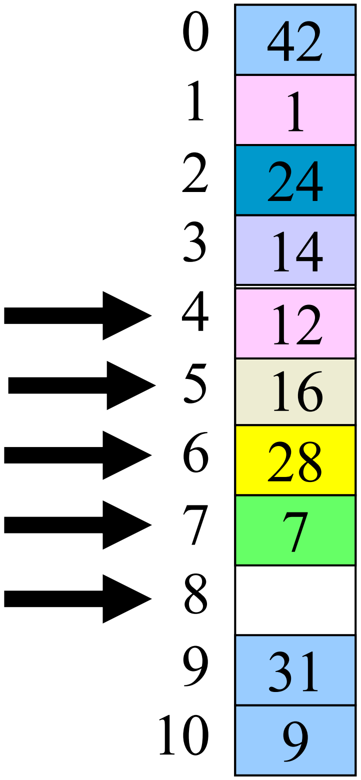


0	42
1	1
2	24
3	14
4	
5	16
6	28
7	7
8	
9	31
10	9

0	42
1	1
2	24
3	14
4	12
5	16
6	28
7	7
8	
9	31
10	9

Search with linear probing (Search 15)

15 = 1 x 11 + 4
15 mod 11 = 4



0	42
1	1
2	24
3	14
4	12
5	16
6	28
7	7
8	
9	31
10	9

NOT FOUND !

Search with linear probing

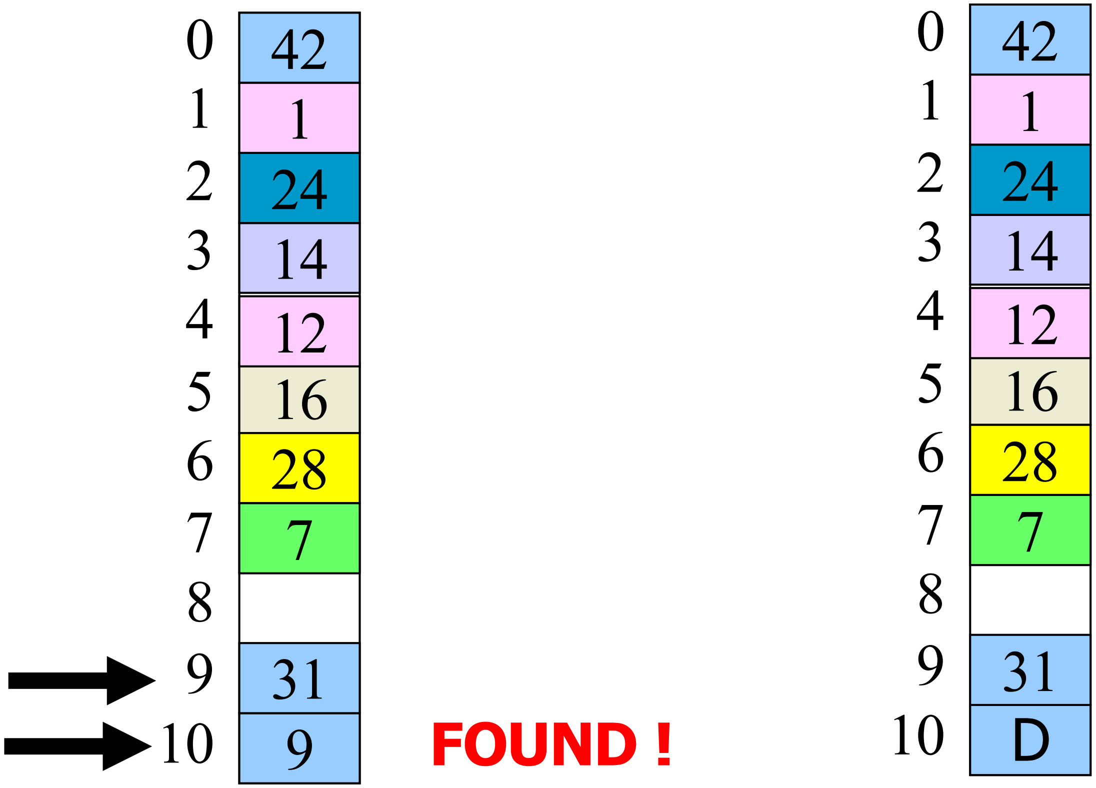
```
// find the slot where searched item should be in

int HashTable<E,K>::hSearch(const K& k) const
{
   int HashVal = k % D;
   int j = HashVal;
    do {// don’ t search past the first empty slot (insert should put it there)
      if (empty[j] || ht[j] == k) return j;
      j = (j + 1) % D;
   } while (j != HashVal);
   return j;  // no empty slot and no match either, give up
}


bool HashTable<E,K>::find(const K& k, E& e) const
{
    int b = hSearch(k);
    if (empty[b] || ht[b] != k) return false;
    e = ht[b];
    return true;
}
```

Deletion in Hashing with Linear Probing

- Since empty buckets are used to terminate search, standard deletion does not work.
- One simple idea is to not delete, but mark.
- Insert: put item in first empty or marked bucket.
- Search: Continue past marked buckets.
- Delete: just mark the bucket as deleted.
- Advantage: Easy and correct.
- Disadvantage: table can become full with dead items.

Deletion with linear probing: LAZY (Delete 9)

9 = 0 x 11 + 9
9 mod 11 = 9



0	42
1	1
2	24
3	14
4	12
5	16
6	28
7	7
8	
9	31
10	9

FOUND !

0	42
1	1
2	24
3	14
4	12
5	16
6	28
7	7
8	
9	31
10	D

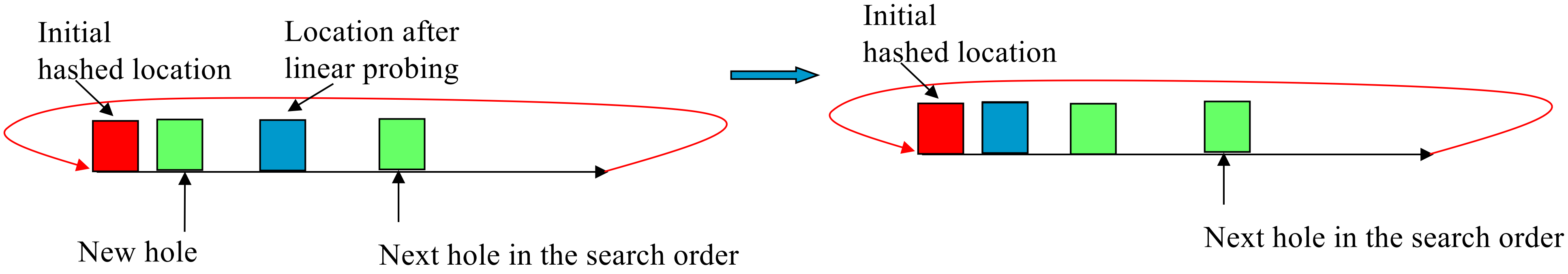
Eager Deletion: fill holes

- Remove and find replacement:
    - □ Fill in the hole for later searches

```
remove(j)
{ i = j;
  empty[i] = true;
  i = (i + 1) % D; // candidate for swapping
  while ((not empty[i]) and i!=j) {
    r = Hash(ht[i]); // where should it go without
collision?
    // can we still find it based on the rehashing strategy?
    if not ((j<r<=i) or (i<j<r) or (r<=i<j))
    then break; // yes find it from rehashing, swap
    i = (i + 1) % D; // no, cannot find it from rehashing
  }
  if (i!=j and not empty[i])
  then {
    ht[j] = ht[i];
    remove(i);
  }
}
```

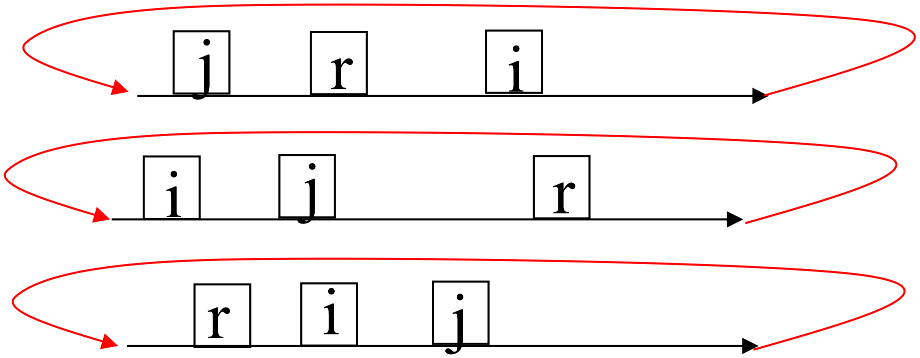
Eager Deletion Analysis (cont.)

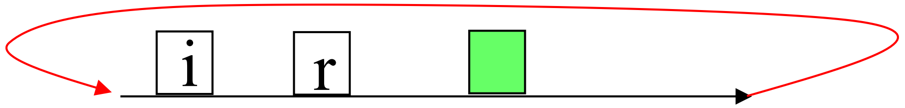
- If not full
  - After deletion, there will be at least two holes
  - Elements that are affected by the new hole are
    - Initial hashed location is cyclically before the new hole
    - Location after linear probing is in between the new hole and the next hole in the search order
    - Elements are movable to fill the hole



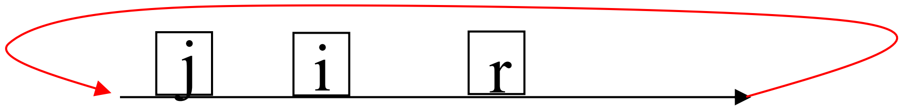
Eager Deletion Analysis (cont.)

- The important thing is to make sure that if a replacement (*i*) is swapped into deleted (*j*), we can still find that element. How can we *not* find it?
  - If the original hashed position (*r*) is circularly in between deleted and the replacement

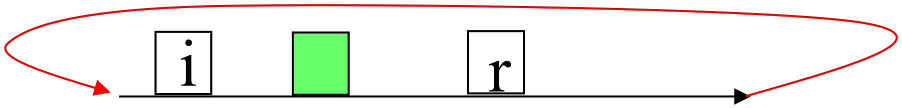




Will not find *i* past the empty green slot!



Will find *i*

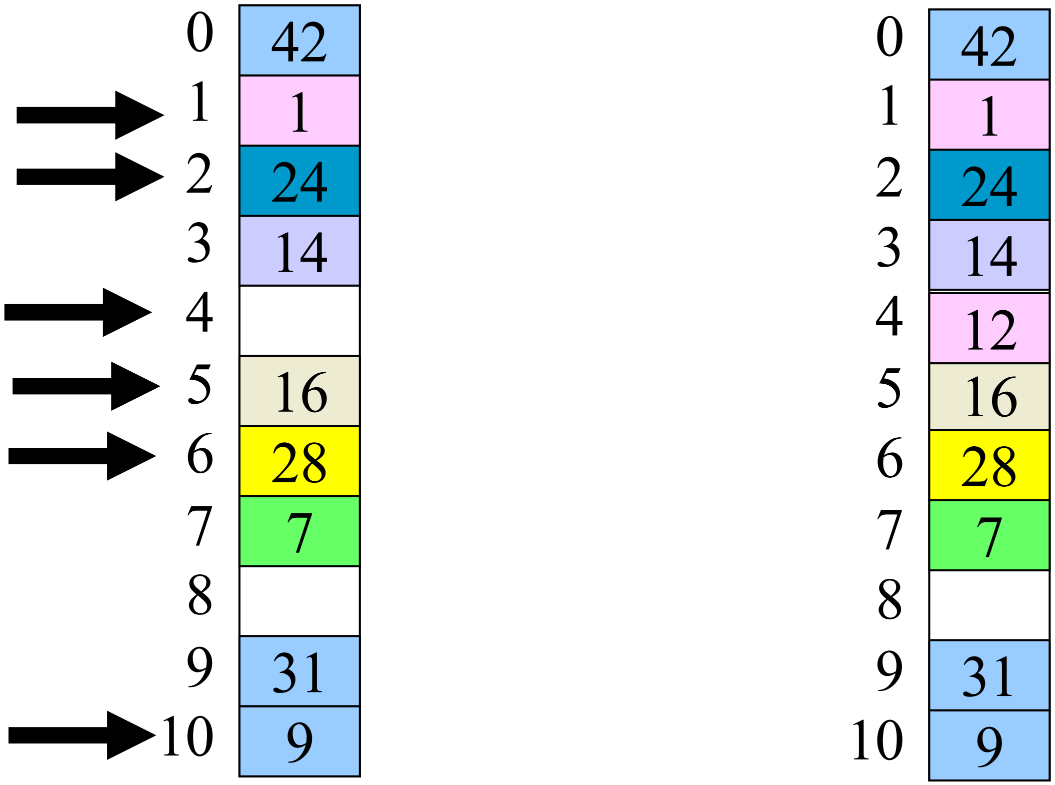


Quadratic Probing

- *Solves the clustering problem in Linear Probing*
  - Check H(x)
  - If collision occurs check **H(x) + 1**
  - If collision occurs check **H(x) + 4**
  - If collision occurs check **H(x) + 9**
  - If collision occurs check **H(x) + 16**
  - ...
  - **H(x) + i^2**

Quadratic Probing (insert 12)

12 = 1 x 11 + 1
12 mod 11 = 1



	0	42
→	1	1
→	2	24
	3	14
→	4	
→	5	16
→	6	28
	7	7
	8	
	9	31
→	10	9

0	42
1	1
2	24
3	14
4	12
5	16
6	28
7	7
8	
9	31
10	9

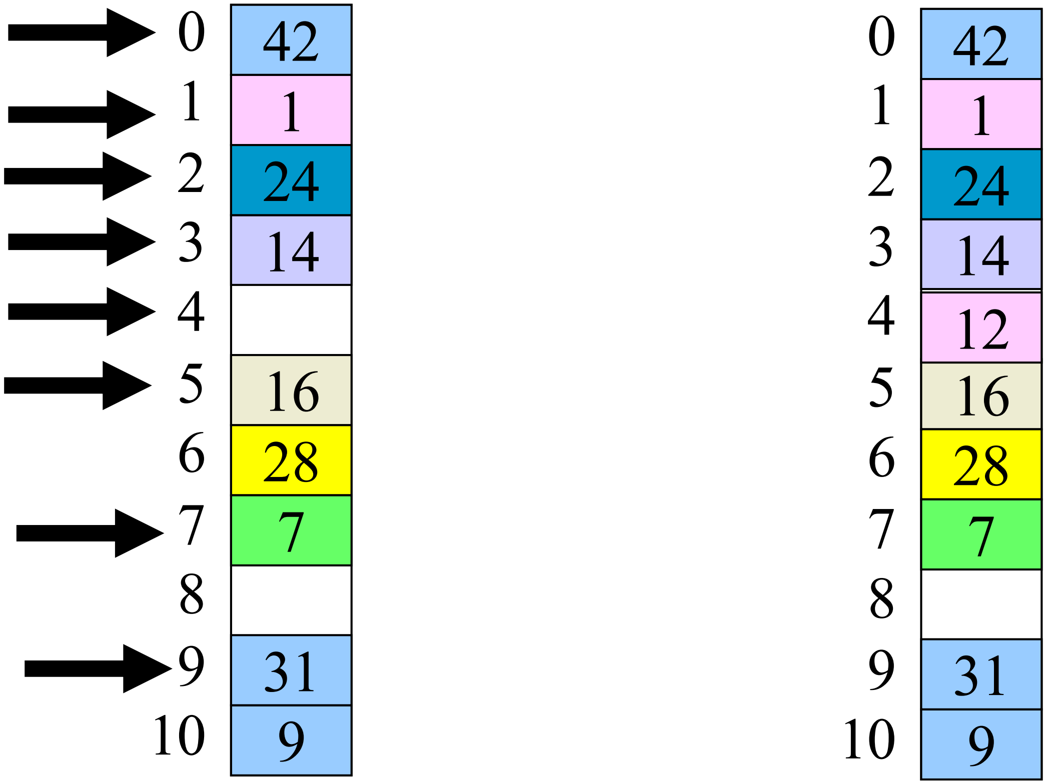
Double Hashing

- *When collision occurs use a second hash function*
  - □ $Hash_2$ (x) = R – (x mod R)
  - □ R: greatest prime number smaller than table-size
- *Inserting 12*

  $H_2(x) = 7 - (x \bmod 7) = 7 - (12 \bmod 7) = 2$
  - □ Check **H(x)**
  - □ If collision occurs check **H(x) + 2**
  - □ If collision occurs check **H(x) + 4**
  - □ If collision occurs check **H(x) + 6**
  - □ If collision occurs check **H(x) + 8**
  - □ **H(x) + i * H_2(x)**

Double Hashing (insert 12)

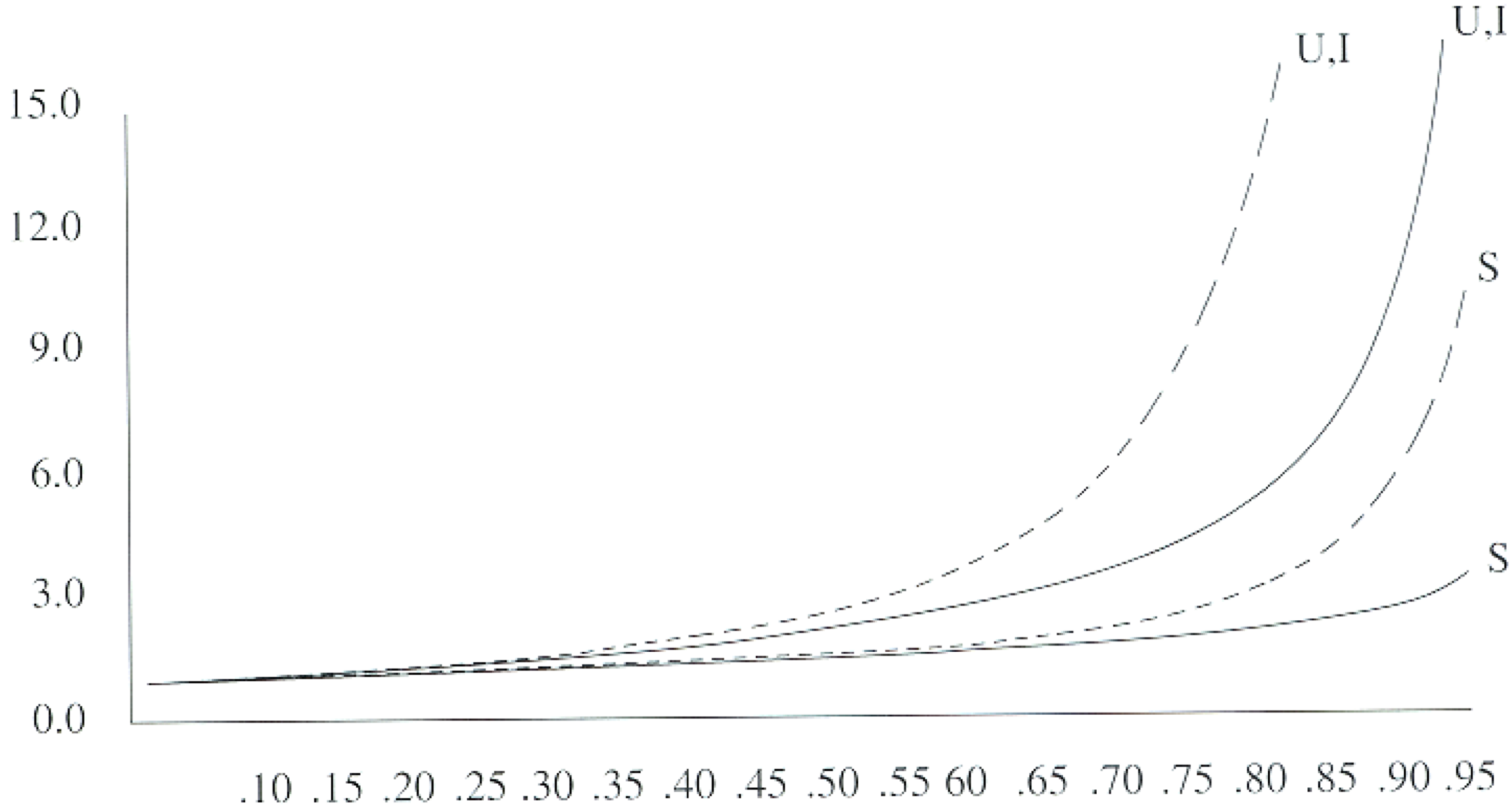
12 = 1 x 11 + 1
12 mod 11 = 1
7 –12 mod 7 = 2



0	42
1	1
2	24
3	14
4	
5	16
6	28
7	7
8	
9	31
10	9

0	42
1	1
2	24
3	14
4	12
5	16
6	28
7	7
8	
9	31
10	9

Comparison of linear and random probing



Rehashing

- If table gets too full, operations will take too long.
- Build another table, twice as big (and prime).
  - Next prime number after 11 x 2 is 23
- Insert every element again to this table

- Rehash after a percentage of the table becomes full (70% for example)

Good and Bad Hashing Functions

- Hash using the wrong key
  - Age of a student
- Hash using limited information
  - First letter of last names (a lot of A's, few Z's)
- Hash functions choices :
  - keys evenly distributed in the hash table
- Even distribution guaranteed by "randomness"
  - No expectation of outcomes
  - Cannot design input patterns to defeat randomness

Examples of Hashing Function

- B=100, N=100, keys = A0, A1, …, A99
- Hashing(A12) = (Ascii(A)+Ascii(1)+Ascii(2)) / B
  - □ H(A18)=H(A27)=H(A36)=H(A45) …
  - □ Theoretically, N(1+N/B)= 200
  - □ In reality, 395 steps are needed because of collision
- How to fix it?
  - □ Hashing(A12) = (Ascii(A)*2^2+Ascii(1)*2+Ascci(2))/B
  - □ H(A12)!=H(A21)
- Examples: numerical keys
  - □ Use X^2 and take middle numbers

Collision Functions

- $H_i(x) = (H(x)+i) \bmod B$
  - □ Linear pobing
- $H_i(x) = (H(x)+ci) \bmod B$ $(c>1)$
  - □ Linear probing with step-size = c
- $H_i(x) = (H(x)+i^2) \bmod B$
  - □ Quadratic probing
- $H_i(x) = (H(x) + i * H_2(x)) \bmod B$

Analysis of Open Hashing

- Effort of one Insert?
  - Intuitively – that depends on how full the hash is
- Effort of an average Insert?
- Effort to fill the Bucket to a certain capacity?
  - Intuitively – accumulated efforts in inserts
- Effort to search an item (both *successful* and *unsuccessful*)?
- Effort to delete an item (both *successful* and *unsuccessful*)?
  - Same effort for successful search and delete?
  - Same effort for unsuccessful search and delete?

More on hashing

- Extensible hashing
  - Hash table grows and shrinks, similar to B-trees

Issues:

- *What do we lose?*
  - **Operations that require ordering are inefficient**
  - **FindMax**: O(n) O(log n) Balanced binary tree
  - **FindMin**: O(n) O(log n) Balanced binary tree
  - **PrintSorted**: O(n log n) O(n) Balanced binary tree
- *What do we gain?*
  - **Insert**: O(1) O(log n) Balanced binary tree
  - **Delete**: O(1) O(log n) Balanced binary tree
  - **Find**: O(1) O(log n) Balanced binary tree
- *How to handle Collision?*
  - Separate chaining
  - Open addressing