Hashing

Dictionary ADTs

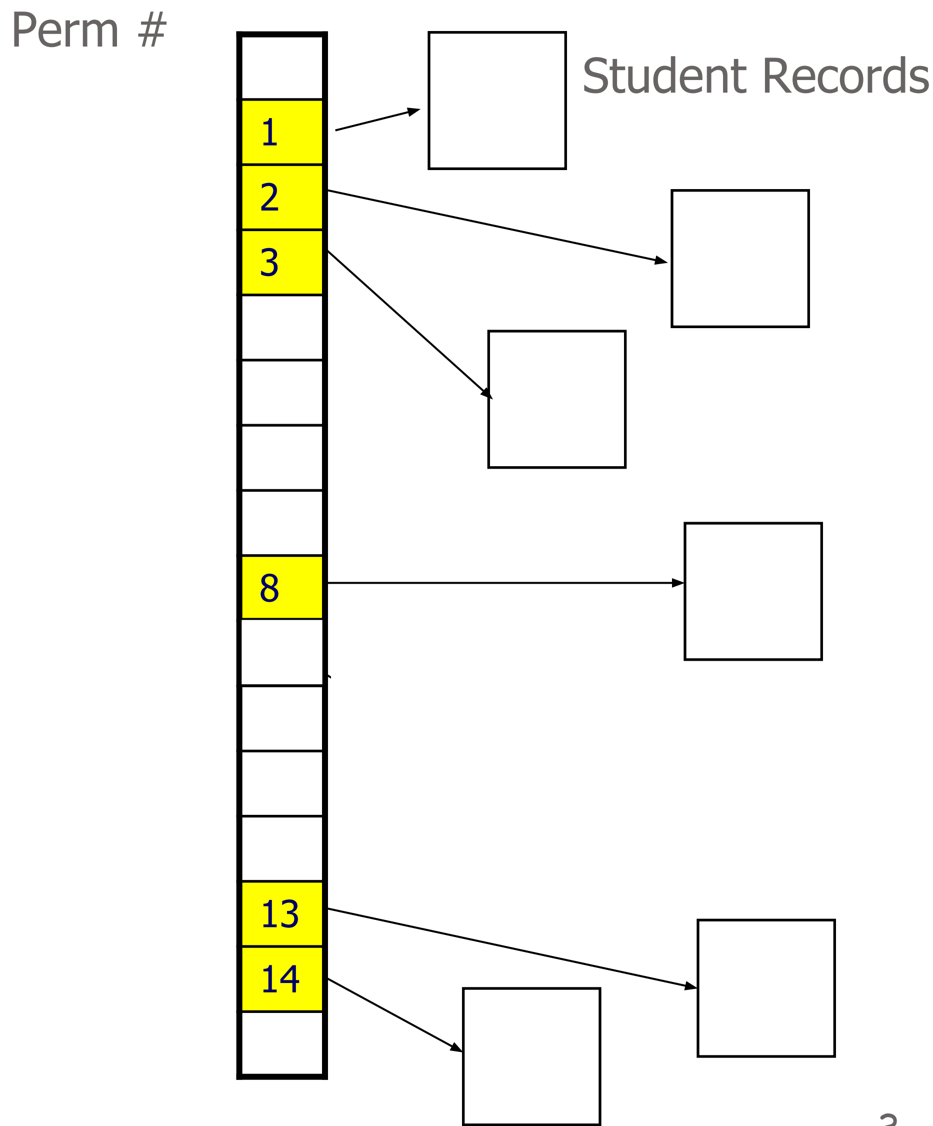
- Data structure with just 3 basic operations:
  - find (i): find item with key i
  - insert (i): insert i into the dictionary
  - remove (i): delete i
  - Just like words in a Dictionary
- Where do we use them:
  - Symbol tables for compiler
  - Customer records (access by name)
  - Games (positions, configurations)
  - Spell checkers
  - P2P systems (access songs by name), etc.

Naïve Method: Linked List

- Keep a linked list of the keys
  - insert (i): add to the head of list. Easy and fast O(1)
  - find (i): worst-case, search the whole list (linear)
  - remove (i): also linear in worst-case

Another Naïve Method: Direct Mapping

- Maintain an array (bit vector) for all possible keys
  - insert (i): set A[i] = 1
  - find (i): return A[i]
  - remove (i): set A[i] = 0



Another Naïve Method: Direct Mapping

- Maintain an array (bit vector) for all possible keys

  - insert (i): set A[i] = 1
  - find (i): return A[i]
  - remove (i): set A[i] = 0

- All operations easy and fast O(1)
- What's the drawback?
- Too much memory/space, and wasteful!

- The space of all possible IP addresses, variable names in a compiler is enormous!

Dictionary ADT: Naïve Implementations

- space-inefficient.
- Linked list space-efficient, but search-inefficient.
- A sorted array does not help, even with ordered keys. The search becomes fast, but insert/delete take linear time.
- Balanced search trees work but take O(log n) time per operation, and complicated.

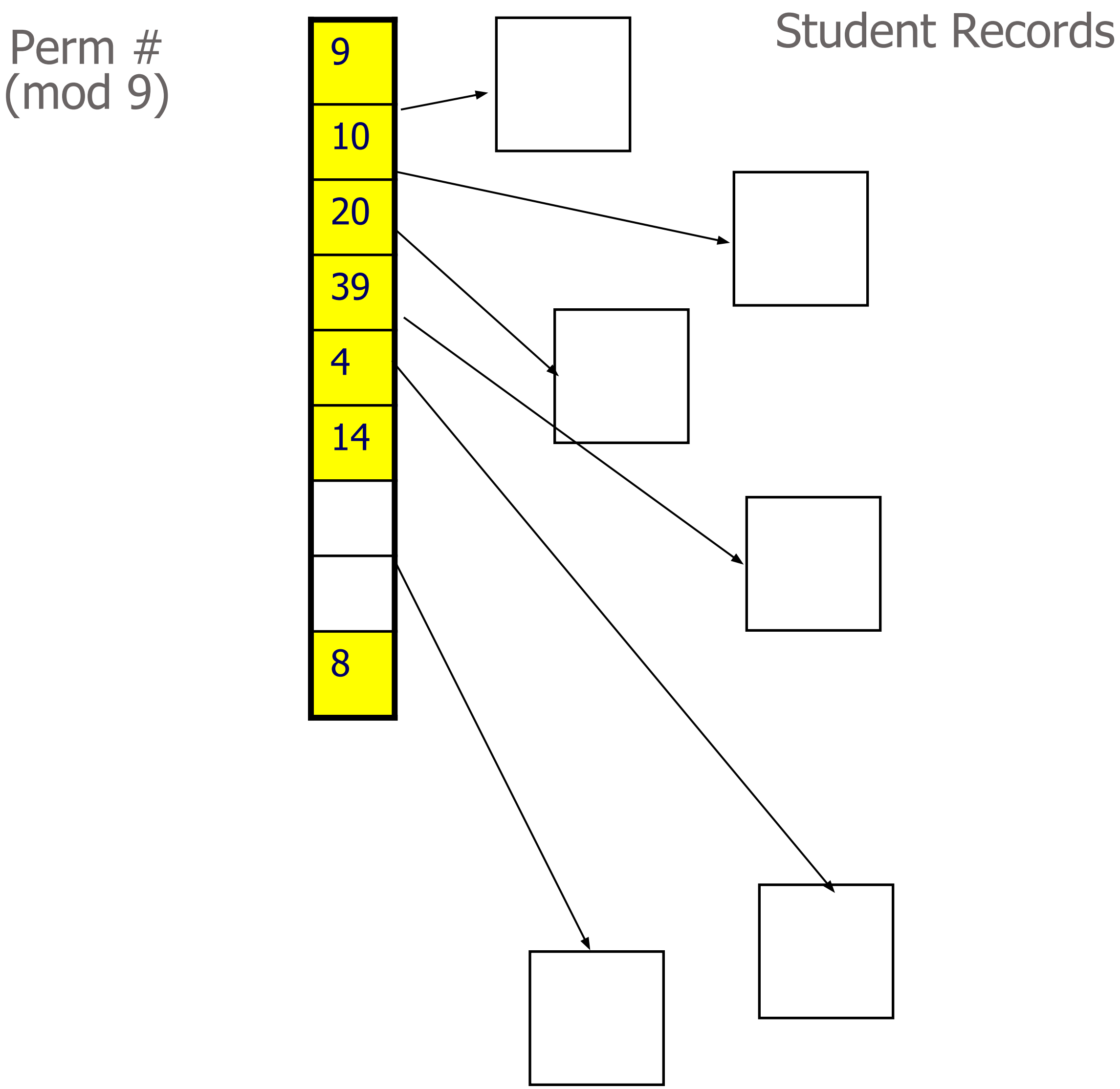
Towards an Efficient Data Structure: Hash Table

- Formal Setup
  - The keys to be managed come from a known but very large set, called universe U
  - We can assume keys are integers {0, 1, …, |U|}
  - Non-numeric keys (strings, webpages) converted to numbers: Sum of ASCII values, first three characters
- The set of keys to be managed is S, a subset of U.
- The size of S is much smaller than U, namely, |S| << |U|
- We use n for |S|.

Hash Table

- Hash Tables use a Hash Function h to map each input key to a unique location in table of size M
  - $h : U \rightarrow \{0, 1, \ldots, M-1\}$
  - hash function determines the hash table size.
- Desiderata:
  - M should be small, O(n)
  - h should be easy to compute
  - Typical example: $h(i) = i \bmod M$

Hashing : the basic idea



Hash Tables: Intuition

- Unique location lets us find an item in O(1) time.
  - Each item is uniquely identified by a key
- Just check the location h(key) to find the item
- What can go wrong?

- Suppose we expect to have at most 100 keys in S
  - 91, 2048, 329, 17, 689345, ….

- We create a table of size 100 and use the hash function h(key) = key mod 100

- It is both fast and uses the ideal size table.

Hashing:

- But what if all keys end with 00?
  - All keys will map to the same location
  - This is called a Collision in Hashing

- This motivates the 3rd important property of hashing
  - A good hash function should evenly spread the keys to foil any special structure of input
  - Hashing with mod 100 works fine if keys random
  - Most data (e.g. program variables) are not random

Hashing:

- A good hash function should evenly spread the keys to foil any special structure of input
- Key idea behind hashing is to “simulate” the randomness through the hash function
- A good choice is $h(x) = x \bmod p$, for prime p
- $h(x) = (ax + b) \bmod p$ called pseudo-random hash functions

Hashing: The Basic Setup

- Choose a pseudo-random hash function h
  - this automatically determines the hash table size.
- An item with key k is put at location h(k).
- To find an item with key k, check location h(k).

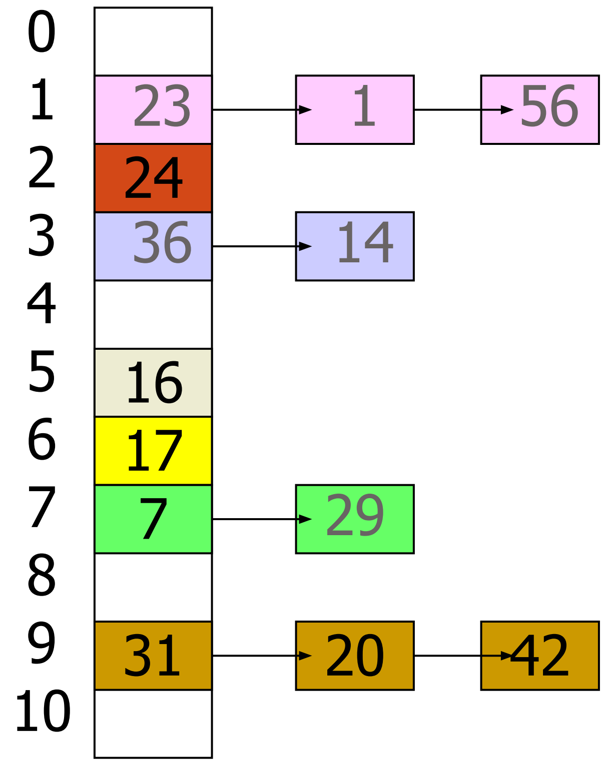
- What to do if more than one keys hash to the same value. This is called collision.
- We will discuss two methods to handle collision:
  - Separate chaining
  - Open addressing

Separate chaining

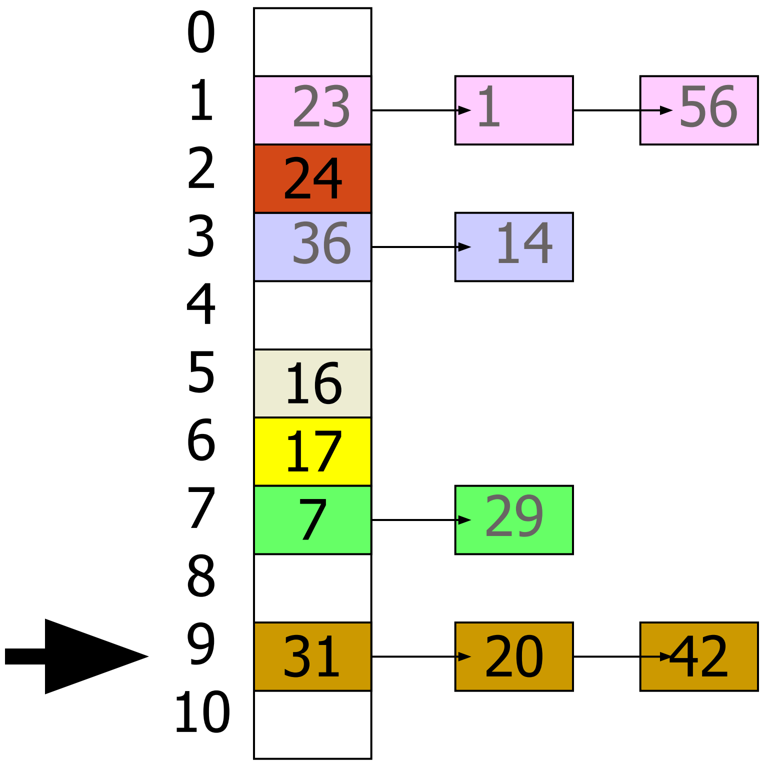
- Maintain a list of all elements that hash to the same value
- Search using the hash function to determine which list to traverse

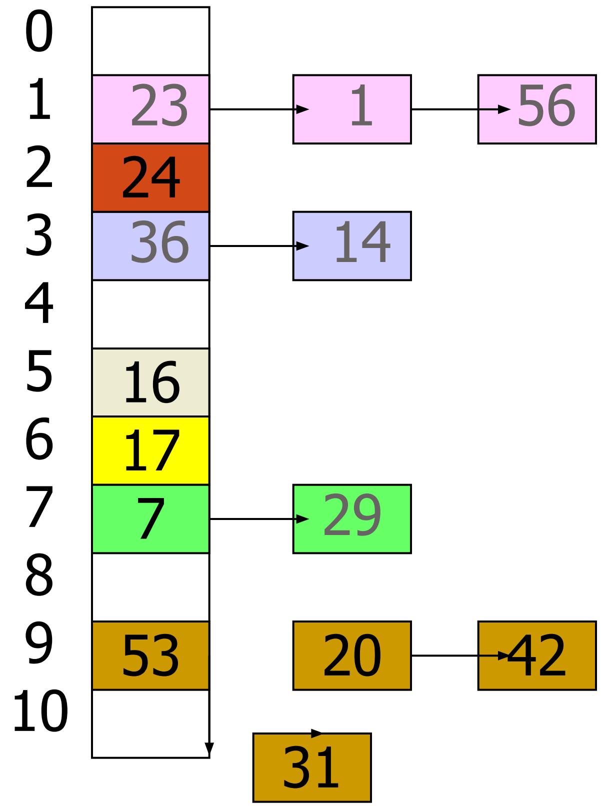
```
find(k,e)
   HashVal = Hash(k,Hsize);
   if
(TheList[HashVal].Search(k,e))
   then return true;
   else return false;
```

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



Insertion: insert 53





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.
  - A worst-case Theorem later
- Average case
  - Keys are uniformly distributed into buckets
  - Load Factor L = InputSize/HashTableSize
  - In a failed search, avg cost is L
  - In a successful search, avg cost is 1 + L/2

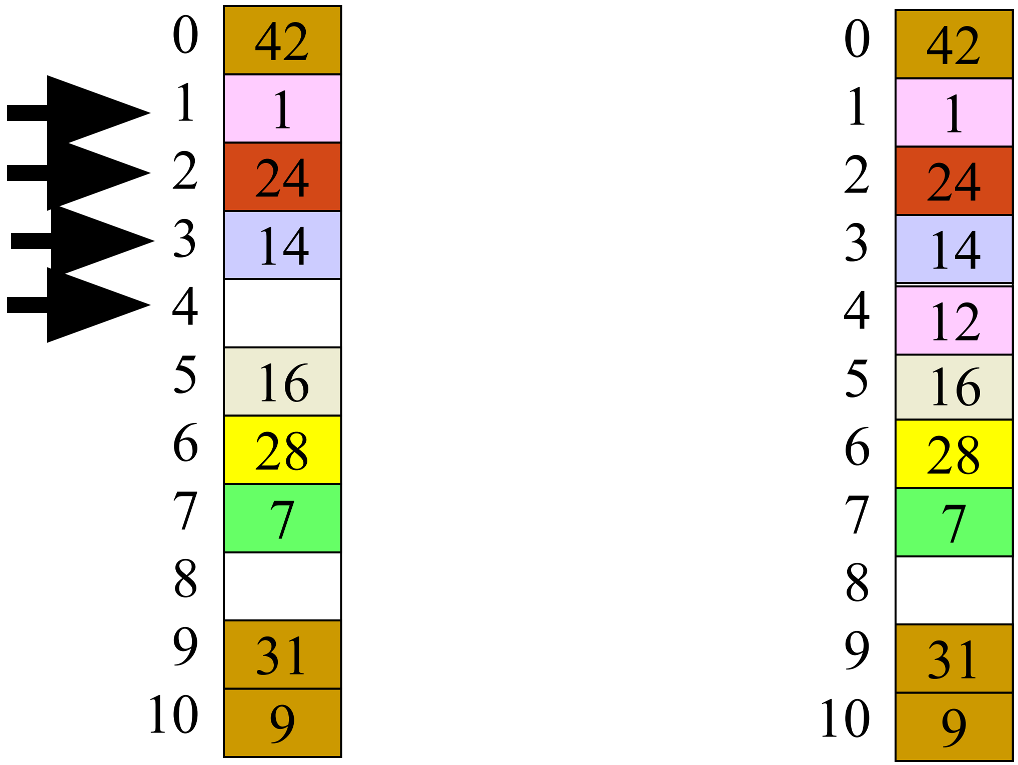
Open addressing

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

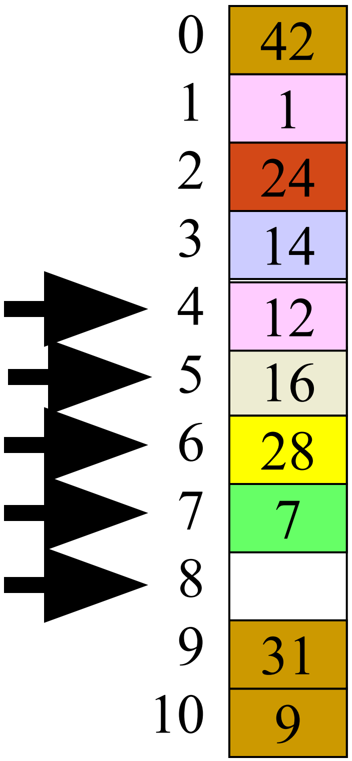
- Linear probing :
  Try next available position

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

Linear Probing (insert 12)



Search with linear probing (Search 15)



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.

- Avg. cost for successful searches $\frac{1}{2}(1 + 1/(1 - L))$
- Failed search avg. cost more $\frac{1}{2}(1 + 1/(1 - L)^2)$

Deletion with linear probing: (Delete 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

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)

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

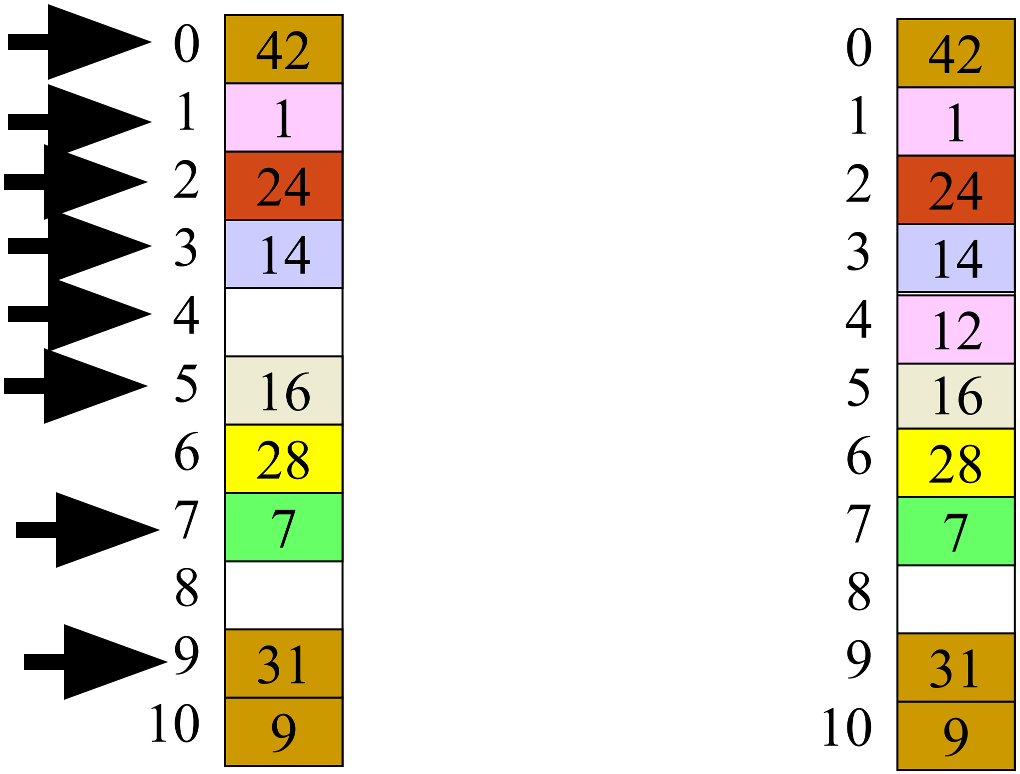
Double Hashing

- *When collision occurs use a second hash function*
  - $Hash_2(x) = R - (x \bmod R)$
  - 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**
  - $\mathbf{H(x) + i * H_2(x)}$

Double Hashing (insert 12)

12 mod 11 = 1
7 −12 mod 7 = 2



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)

Collision Functions

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