Skip List and Hashing

Chapter 10

Dictionary

A **dictionary** is a collection of pairs of the form (k, v), where k is a **key** and v is the **value** associated with the key k (equivalently, v is the value whose key is k). No two pairs in a dictionary have the same key.

The following operations are performed on a dictionary:

- Determine whether or not the dictionary is empty.
- Determine the dictionary size (i.e., number of pairs).
- Find the pair with a specified key.
- Insert a pair into the dictionary.
- Delete or erase the pair with a specified key.

ADT - Dictionary

AbstractDataType *Dictionary*
{
instances
collection of pairs with distinct keys

operations
empty() : return true iff the dictionary is empty;

size() : return the number of pairs in the dictionary;

find(k) : return the pair with key k;

insert(p) : insert the pair p into the dictionary;

erase(k) : delete the pair with key k;
}

Application of Dictionary

- Collection of student records in a class
  - (key, value) =
    (student-number, a list of assignment and exam marks)
  - All keys are distinct
- Get the element whose key is Tiger Woods
- Update the element whose key is Seri Pak

- Dictionary is used to implement **SKIP LISTS**

Skip Lists

- The search operation on a sorted array using the binary search method takes **O(log*n*)**
- The search operation on a sorted chain takes **O(*n*)**
- How can we improve the search performance of a sorted chain?

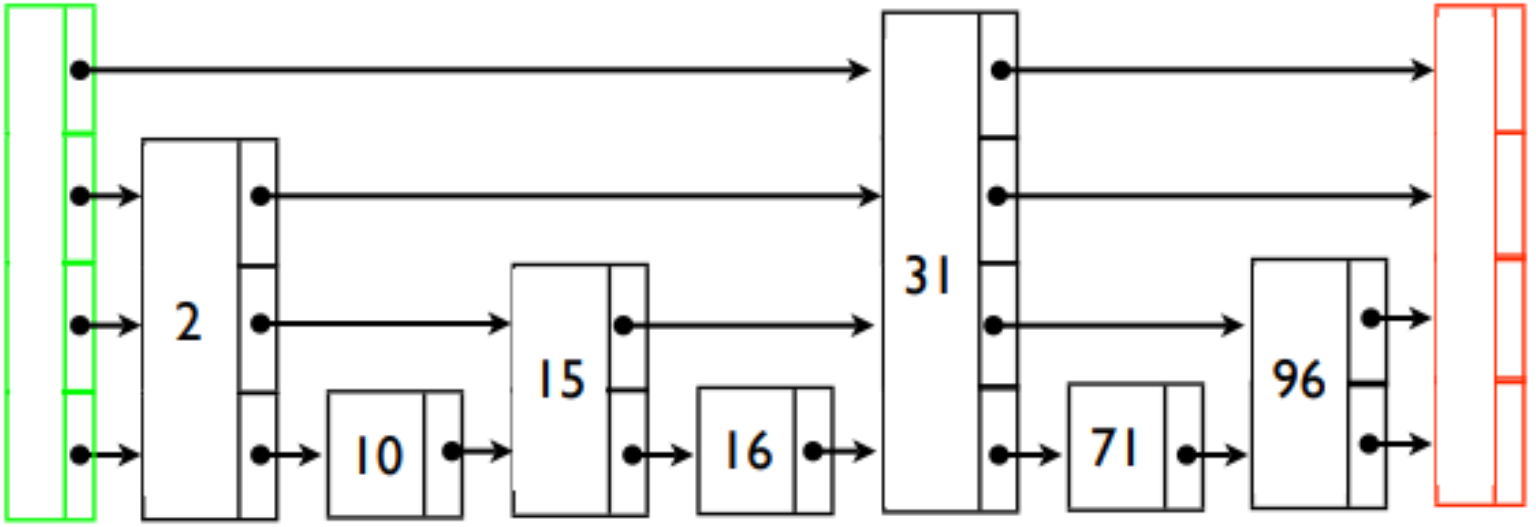
  → **By putting additional pointers in some of the chain nodes**

- Chains augmented with additional forward pointers are called **skip lists**

Skip List

- A skip list is a collection of lists at different *levels*
- The lowest level (0) is a sorted, singly linked list of all nodes
- The first level (1) links alternate nodes
- The second level (2) links every fourth node
- In general, level i links every 2^i*th* node
- In total, $\lceil \log_2 n \rceil$ levels (i.e. $O(\log_2 n)$ levels).
- Each level has half the nodes of the one below it

Skip List

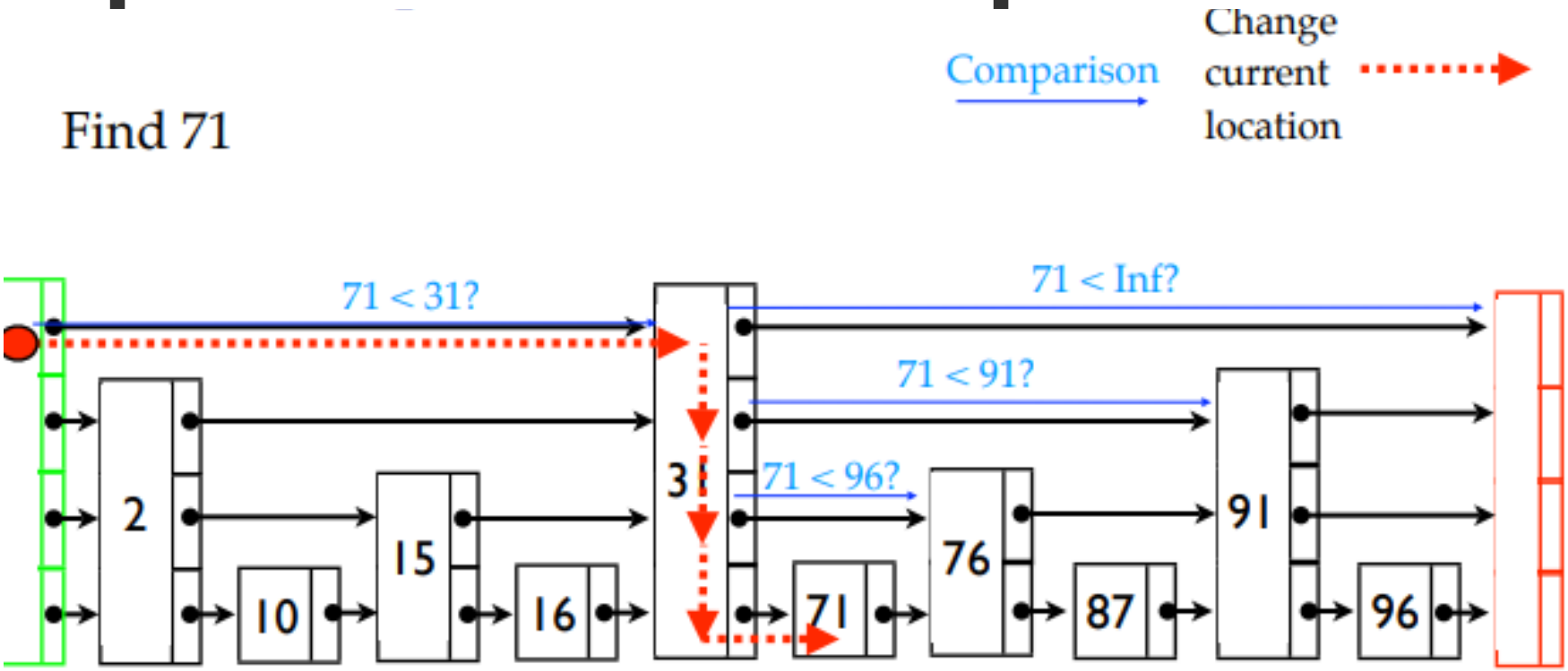


- Keys in sorted order.
- O(log *n*) *levels*
- Each higher level contains 1/2 the elements of the level below it.
- Header & sentinel nodes are in every level

Skip List Search

- To search for an element with a given key:
  - Find location in top list
    - Top list has O(1) elements with high probability
    - Location in this list defines a range of items in next list
  - Drop down a level and recurse
- O(1) time per level on average
- O(lg *n*) levels with high probability
- Total time: O(lg *n*)

Skip List Search Example



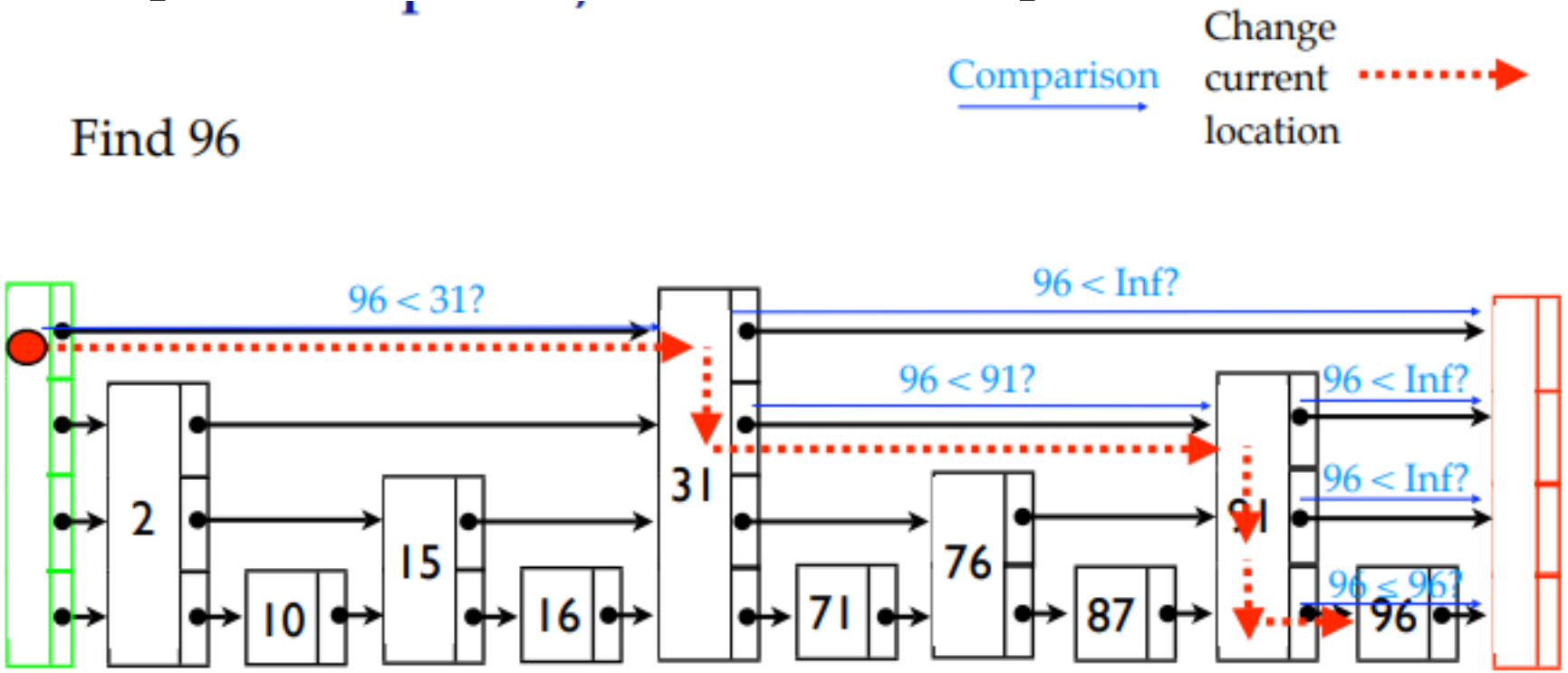
When search for k:

If k = key, done!

If k < next key, go down a level

If k ≥ next key, go right

Skip List Search example



When search for k:

If k = key, done!

If k < next key, go down a level

If k ≥ next key, go right

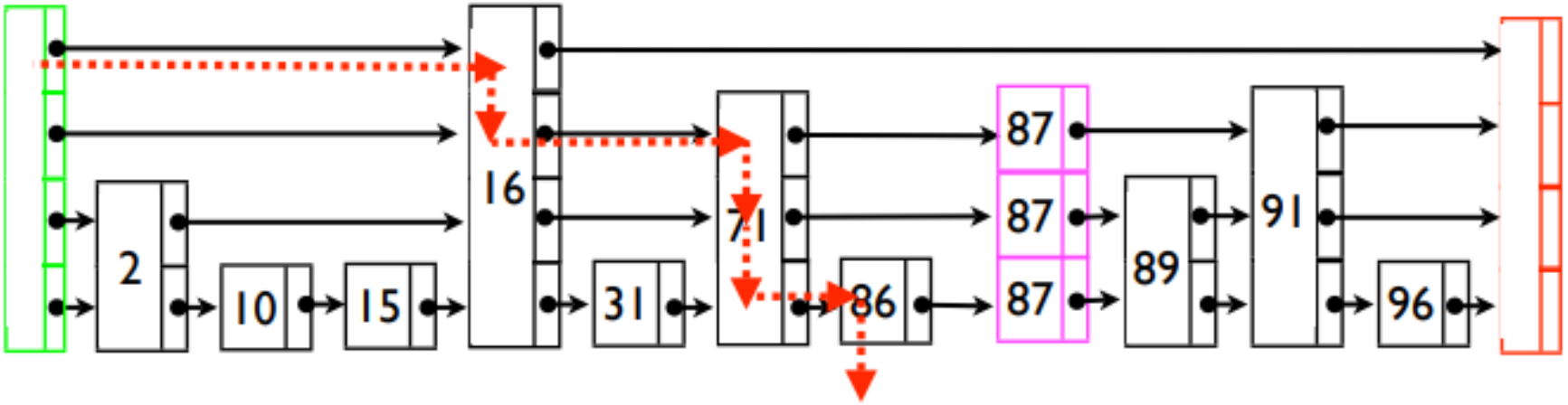
Skip List Insert

- Skip list insert: analysis
  - Do a search for that key
  - Insert element in bottom-level list
  - With probability *p*, recurse to insert in next level
  - Expected number of lists = $1 + p + p^2 + \ldots$

    $= 1/(1-p) = O(1)$ if *p* is constant
  - Total time = Search + $O(1) = O(\lg n)$ expected
- Skip list delete: O(1)

Insertion Example

Insert 87




```
Find k
Insert node in level 0
let i = 1
while FLIP() == "heads":
    insert node into level i
    i++
```

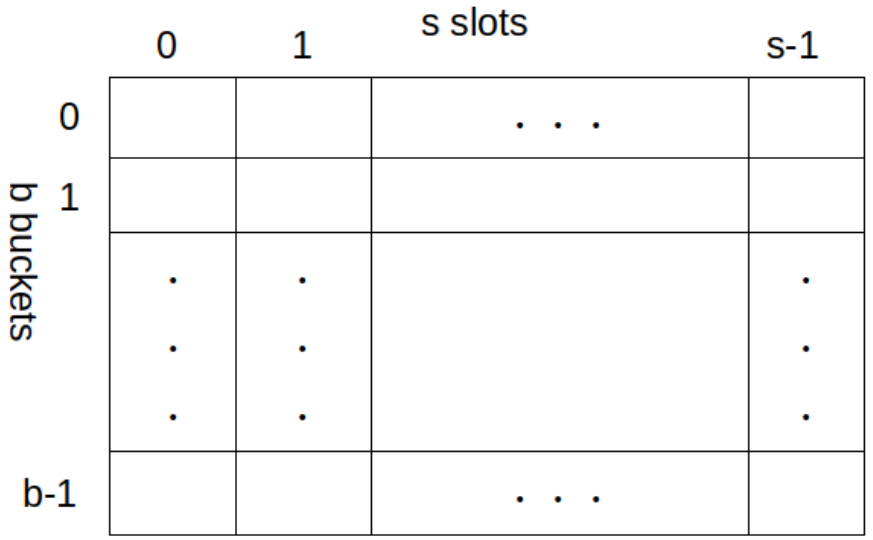
Just insertion into a linked list after last visited node in level *i*

Hash Table

- A **hash table** is an alternative method for representing a dictionary
- In a hash table, a **hash function** is used to map keys into positions in a table. This act is called **hashing**
- The ideal hashing case: if a pair *p* has the key *k* and *f* is the hash function, then *p* is stored in position *f(k)* of the table
- Hash table is used in many real world applications.

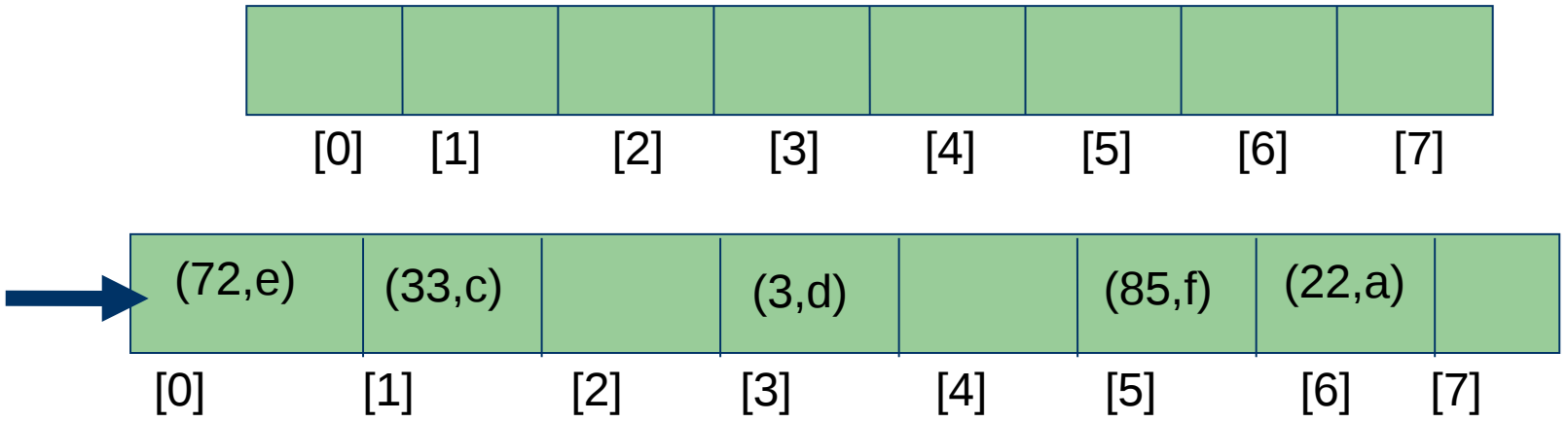
Static Hashing

- Key-value pairs are stored in a fixed size table called a *hash table*.
  - A hash table is partitioned into many ***buckets***.
  - Each bucket has many ***slots***.
  - Each slot holds one record.
  - A hash function f(x) transforms the identifier (key) into an address in the hash table

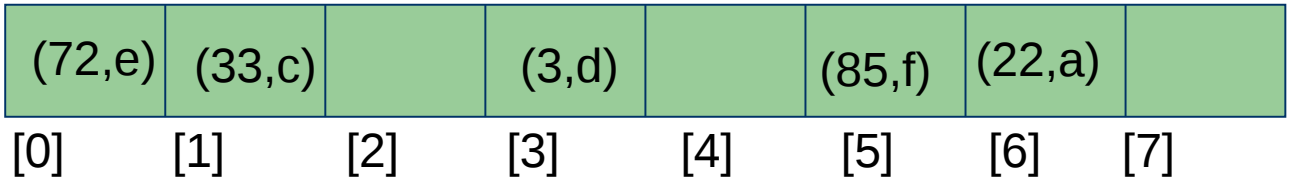


Ideal Hashing

- Pairs are: (22,a),(33,c),(3,d),(72,e),(85,f)
- Hash table is ht[0:7], b = 8 (where b is the number of positions in the hash table)
- Hash function ***f* is key % b** = key % 8
- Where are the pairs stored?



Collision



- Where does (25,g) go?
- The **home bucket** for (25,g) is already occupied by (33,c)
  This situation is called **collision**
- Keys that have the same home bucket are called **synonyms**
  - 25 and 33 are synonyms with respect to the hash function that is in use

Overflow

- A **collision** occurs when the home bucket for a new pair is occupied by a pair with different key
- An **overflow** occurs when there is **no space in the home bucket** for the new pair
- When a bucket can hold only one pair, collisions and overflows occur together
- Need a method to handle overflows

Overflow handling

We may handle overflows by:

- Search the hash table in some systematic fashion for a bucket that is not full.
- Linear probing (linear open addressing).
- Quadratic probing.
- Random probing.

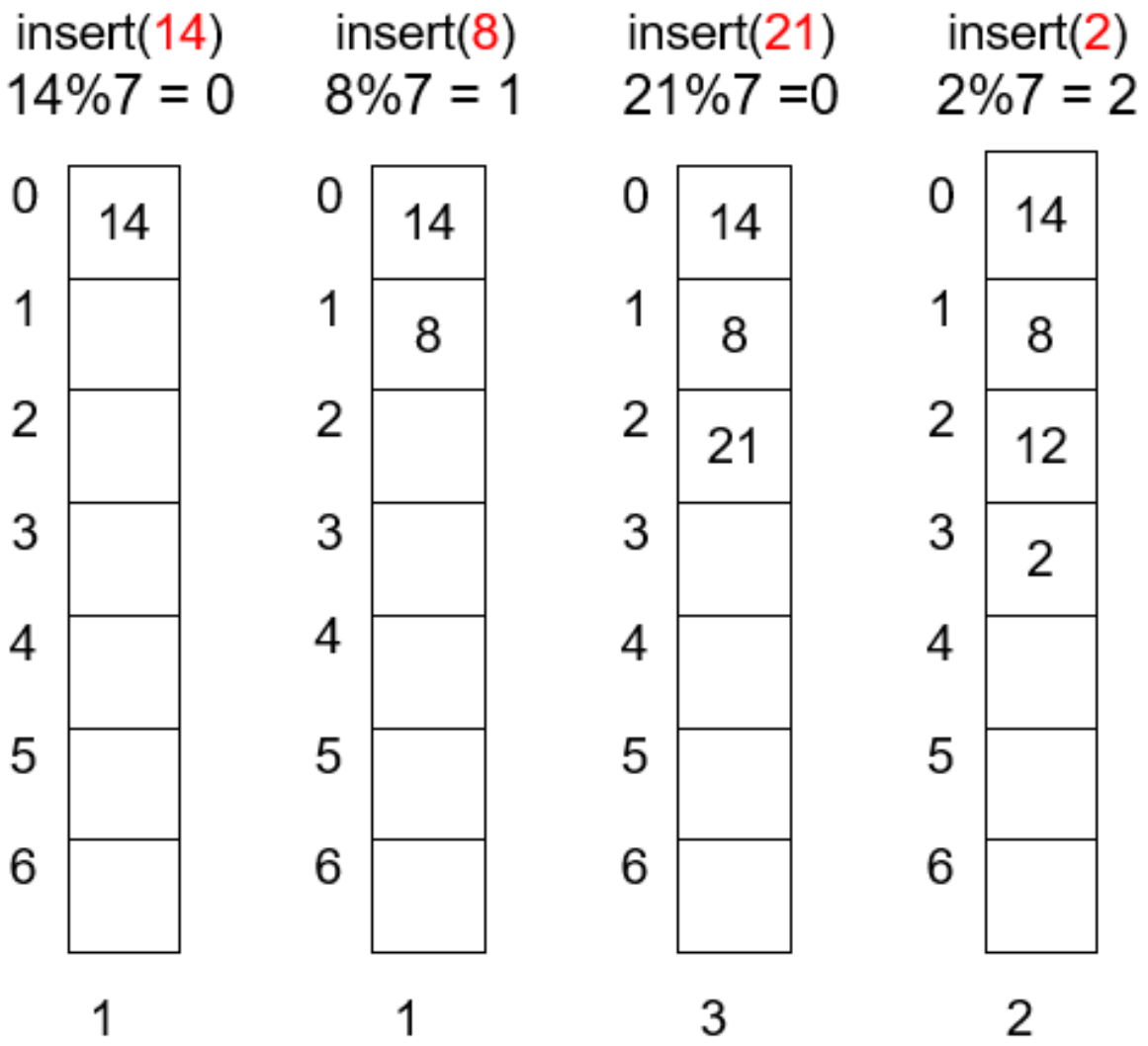
Eliminate overflows by permitting each bucket to keep a list of all pairs for which it is the home bucket.

- Array linear list.
- Chain.

Linear Probing

- Main Idea: When collision occurs, scan down the array one cell at a time looking for an empty cell
  - $h_i(X) = (\text{Hash}(X) + i) \bmod \textit{TableSize} \quad (i = 0, 1, 2, \ldots)$
    - Compute hash value and increment it until a free cell is found

Linear Probing example



- Hash Table Operations
  - **Search**: compute f(k) and see if a pair exists
  - **Insert**: compute f(k) and place it in that position
  - **Delete**: compute f(k) and delete the pair in that position