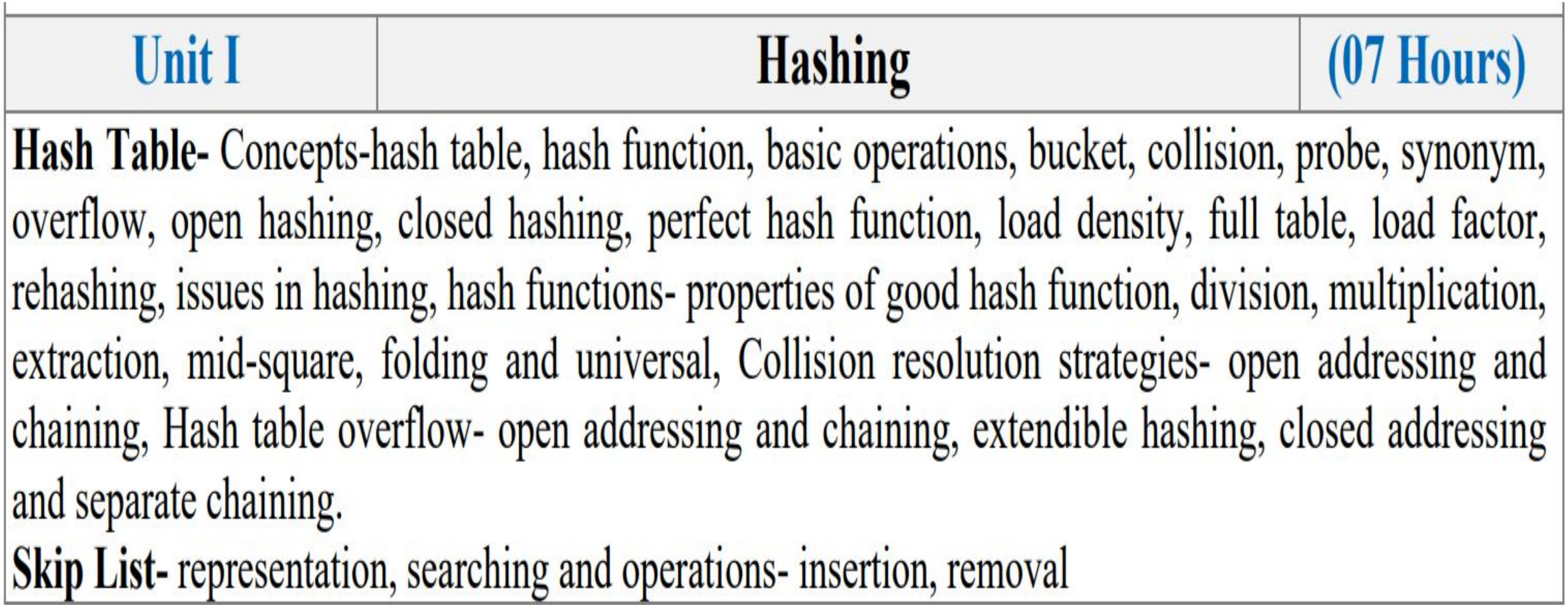
# Unit I



<table>
<tr><th>Unit I</th><th>Hashing</th><th>(07 Hours)</th></tr>
<tr><td colspan="3"><b>Hash Table-</b> Concepts-hash table, hash function, basic operations, bucket, collision, probe, synonym, overflow, open hashing, closed hashing, perfect hash function, load density, full table, load factor, rehashing, issues in hashing, hash functions- properties of good hash function, division, multiplication, extraction, mid-square, folding and universal, Collision resolution strategies- open addressing and chaining, Hash table overflow- open addressing and chaining, extendible hashing, closed addressing and separate chaining.<br><b>Skip List-</b> representation, searching and operations- insertion, removal</td></tr>
</table>

# Why Hashing?

- Internet has grown to millions of users generating **terabytes** of content every day.
- According to internet data tracking services, the amount of content on the internet **doubles every six** months.
- With this kind of growth, it is impossible to find anything in the internet, unless we develop new data structures and algorithms for storing and accessing data.

# So what is wrong with traditional data structures like Arrays and Linked Lists?

- Suppose we have a very large data set stored in an array. The amount of time required to look up an element in the array is either **O(log n)** or **O( n)** based on whether the array is sorted or not.
- If the array is sorted then a technique such as binary search can be used to search the array. Otherwise, the array must be searched linearly.
- Either case may not be desirable if we need to process a very large data set. Therefore we discuss a new technique called **hashing** that allows us to update and retrieve any entry in constant time **O(1)**. The constant time or **O(1)** performance means, the amount of time to perform the operation does not depend on data size n.

# Hash Data Structure

- one of the most widely used data structure after arrays
- supports search, insert and delete in O(1) time on average which is more efficient than other popular data structures like arrays, Linked List
- use hashing for dictionaries, frequency counting, maintaining data for quick access by key
- Real World Applications include Database Indexing, Cryptography, Caches, Symbol Table and Dictionaries
- two forms of hash typically implemented in programming languages.

  **Hash Set** : Collection of unique keys

  **Hash Map** : Collection of key value pairs with keys being unique ( dictionary in Python)

# Introduction to Hashing

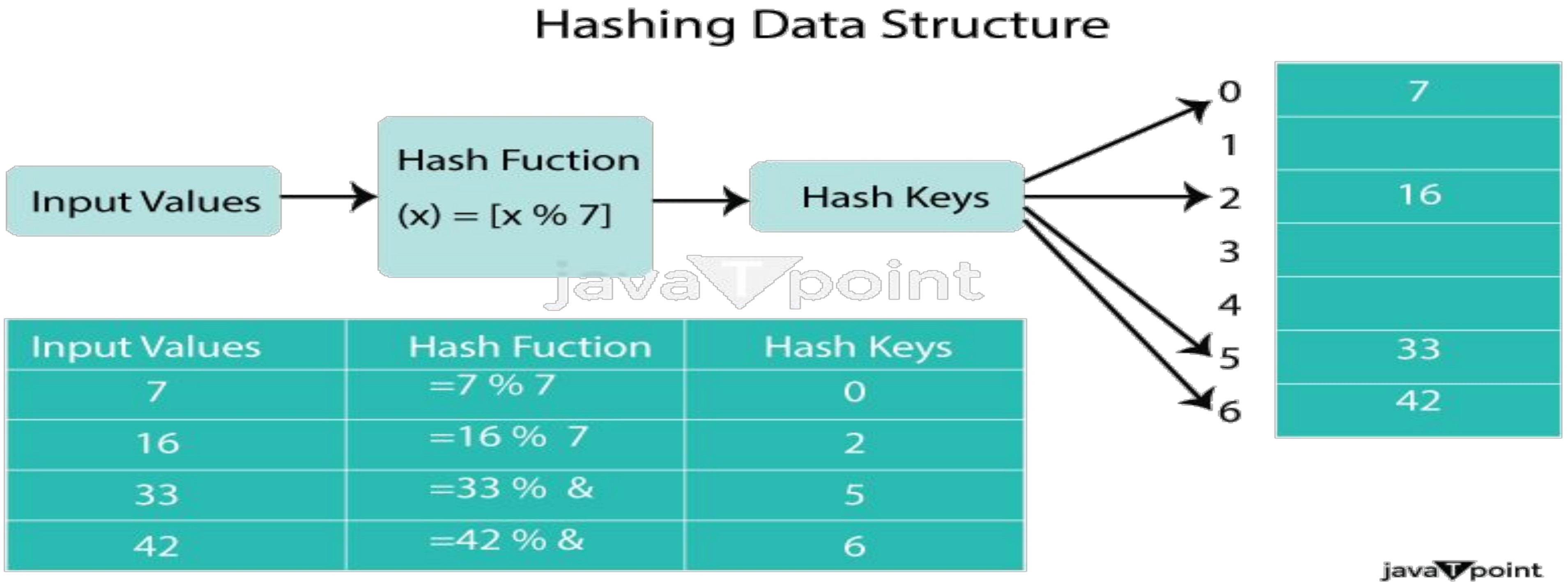
## Hashing

- the process of mapping a key value to a position in a table.
- to create a unique identifier for a piece of data,which can be used to quickly look up that data in a large dataset.
- Involves mapping data to a specific index in a hash table (an array of items) using a hash function that enables fast retrieval of information based on its key

# Introduction to Hashing

A hash function maps key values to positions.

- a type of mathematical operation that takes an input (or key) and outputs a fixed-size result known as a hash code or hash value.
- the hash function should produce a unique hash code for each input, which is known as the hash property.



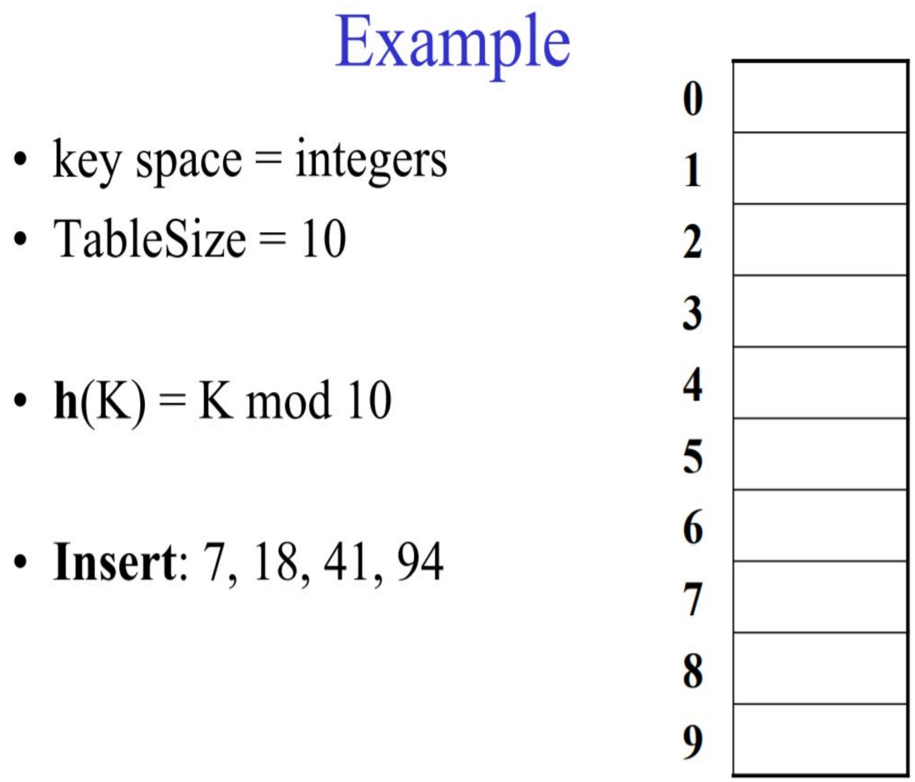
Hashing Data Structure

| Input Values | Hash Fuction | Hash Keys |
|---|---|---|
| 7 | =7 % 7 | 0 |
| 16 | =16 % 7 | 2 |
| 33 | =33 % & | 5 |
| 42 | =42 % & | 6 |

# Hash Function

- **Consider the problem of searching an array for a given value**.
- If the array is **not sorted**, the search might require examining each and all elements of the array.
- If the array is **sorted**, (binary search)
  worse-case runtime complexity to O(log n).
  **Hash function:** O(1)

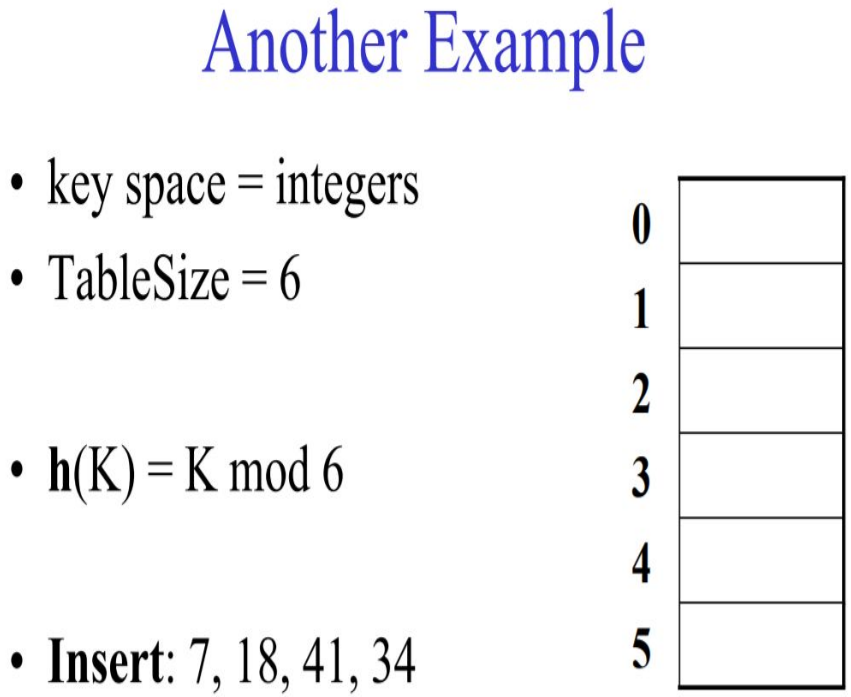
# Examples:



## Example

- key space = integers
- TableSize = 10
- **h**(K) = K mod 10
- **Insert**: 7, 18, 41, 94

| | |
|---|---|
| **0** | |
| **1** | |
| **2** | |
| **3** | |
| **4** | |
| **5** | |
| **6** | |
| **7** | |
| **8** | |
| **9** | |



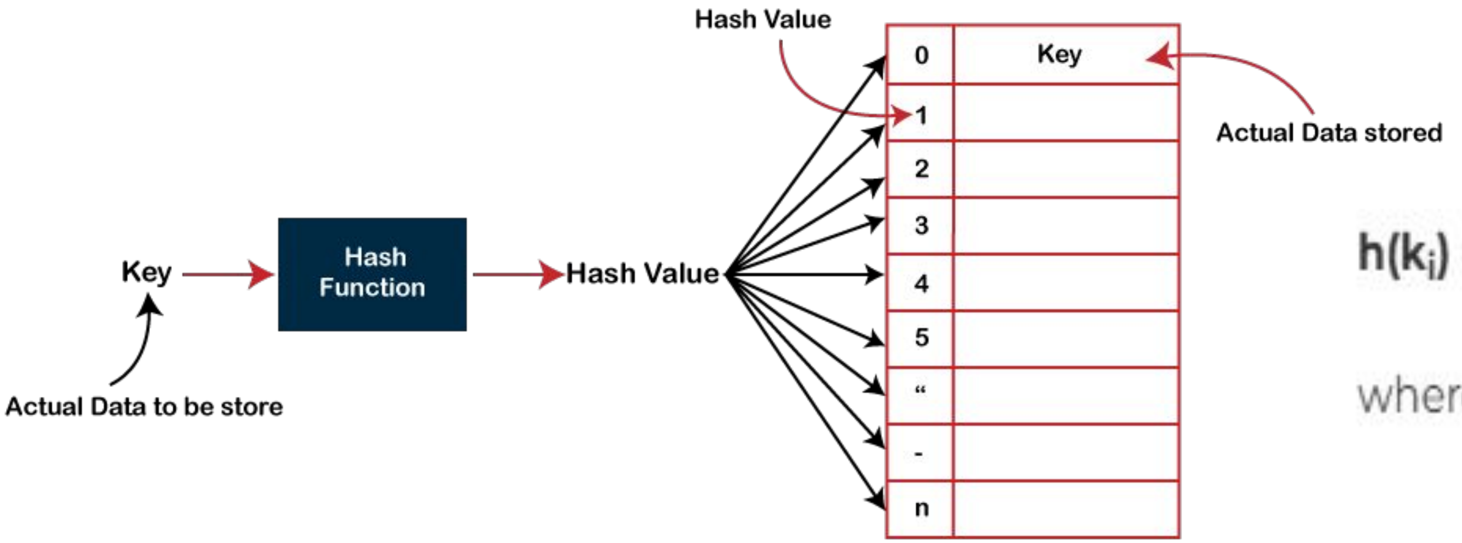
## Another Example

- key space = integers
- TableSize = 6
- **h**(K) = K mod 6
- **Insert**: 7, 18, 41, 34

| | |
|---|---|
| **0** | |
| **1** | |
| **2** | |
| **3** | |
| **4** | |
| **5** | |

# **Hash Table-** Concepts-hash table

**Hash table** (**hash map**) is a data structure that implements an associative array abstract data type, a structure that can map keys to values. A hash table uses a hash function to compute an *index* into an array of *buckets* or *slots*, from which the desired value can be found.



$h(k_i) = k_i \% m;$

where **m** is the size of the hash table.

# Bucket

- a storage unit within a hash table where data elements are stored after being hashed using a hash function

- Overflow: there is no space in the bucket for the new pair.

# Overflow Handling

- An overflow occurs when the home bucket for a new pair (key, element) is full.
- 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.

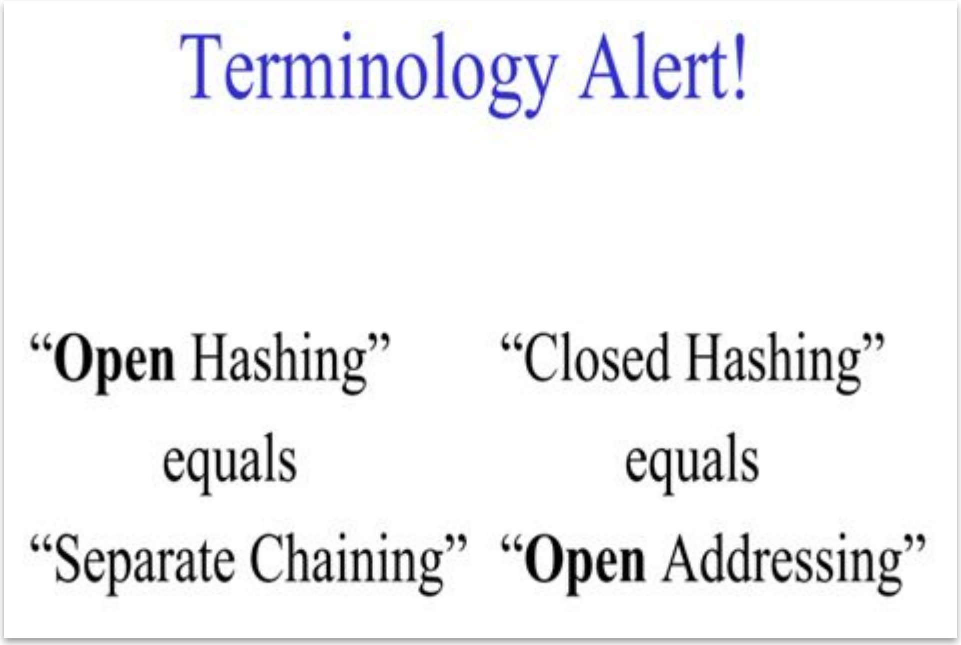
# Collision

## Collision Resolution

**Collision**: when two keys map to the same location in the hash table.

Two ways to resolve collisions:

1. Separate Chaining
2. Open Addressing (linear probing, quadratic probing, double hashing)



### Terminology Alert!

“**Open** Hashing”
equals
“Separate Chaining”

“Closed Hashing”
equals
“**Open** Addressing”

# Probe

- Each calculation of an address and test for success is known as a Probe.

# Synonym

- It is possible for different keys to hash to the same array location. This situation is called collision and the colliding keys are called **synonyms**.

# Overflow

The condition of bucket-**overflow** is known as collision.

This is a fatal state for any static **hash** function.

In this case, **overflow** chaining can be used.

**Overflow** Chaining − When buckets are full, a new bucket is allocated for the same **hash** result and is linked after the previous one.

# Open hashing

- **Open Hashing**
- In open **hashing**, keys are stored in linked lists attached to cells of a **hash** table.

# Closed hashing

- **Closed Hashing** (Open Addressing):
  - In **closed hashing**, all keys are stored in the **hash** table itself without the use of linked lists.

# Perfect hash function

Given a collection of items, a hash function that maps each item into a unique slot is referred to as a **perfect hash function**.

-Avoid collision

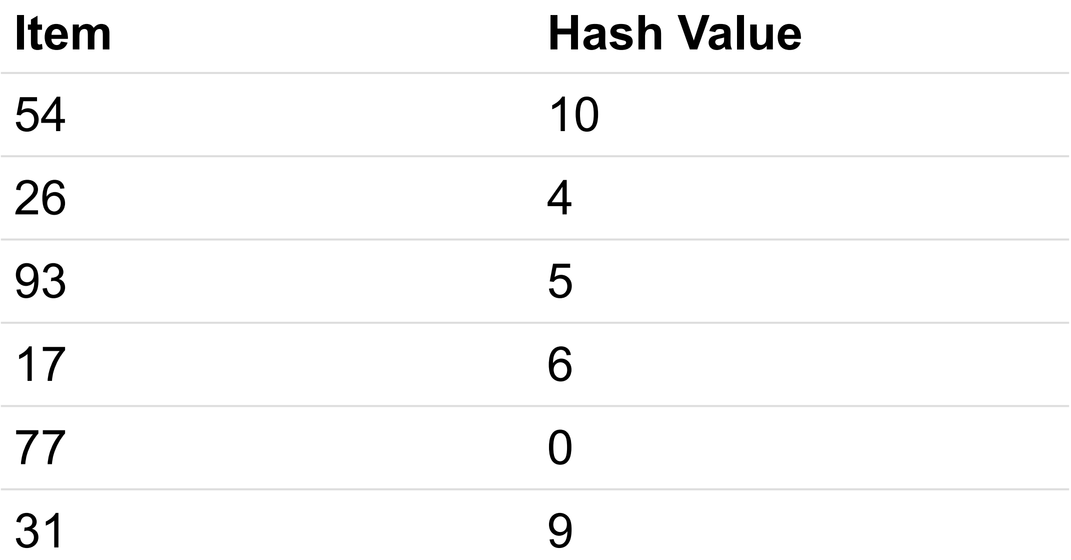
# Full table

- A full table does not have an empty location to store a new record.

# Load density, Load factor:

The load factor of a hash table is the ratio of the number of keys in the table to the size of the hash table

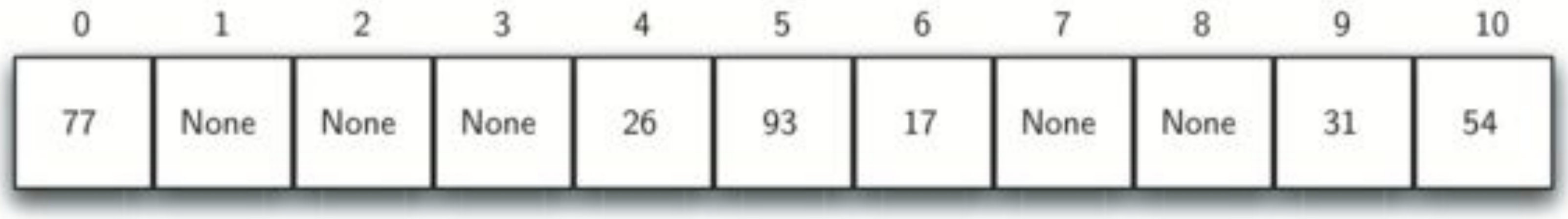
- Assume that we have the set of integer items
- 54, 26, 93, 17, 77, and 31.
- hash function = remainder method □ simply takes an item and divides it by the table size, returning the remainder as its hash value (h(item)=item%11).



| Item | Hash Value |
|---|---|
| 54 | 10 |
| 26 | 4 |
| 93 | 5 |
| 17 | 6 |
| 77 | 0 |
| 31 | 9 |

# load factor (Cont.)

The load factor of a hash table is the ratio of the number of keys in the table to the size of the hash table



| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| 77 | None | None | None | 26 | 93 | 17 | None | None | 31 | 54 |

Note that 6 of the 11 slots are now occupied. This is referred to as the **load factor**, and is commonly denoted by **λ=number of items/table size.**

For this example, **λ=6/11**.

# Load Factor (open addressing)

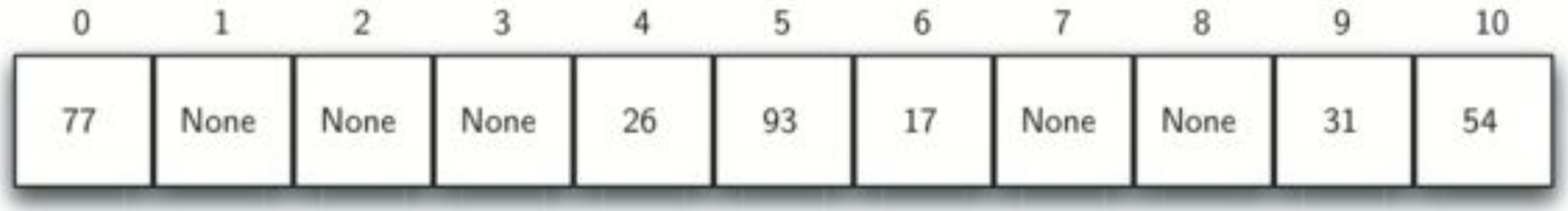
**definition:** The load factor λ of a probing hash table is the fraction of the table that is full.

The **load factor** ranges from 0 (empty) to 1 (completely full).

- It is better to keep the load factor under 0.7
- **Double** the table size and **rehash** if load factor gets high
- Cost of Hash function f(x) must be minimized
- When collisions occur, linear probing can always find an empty cell
  - But clustering can be a problem

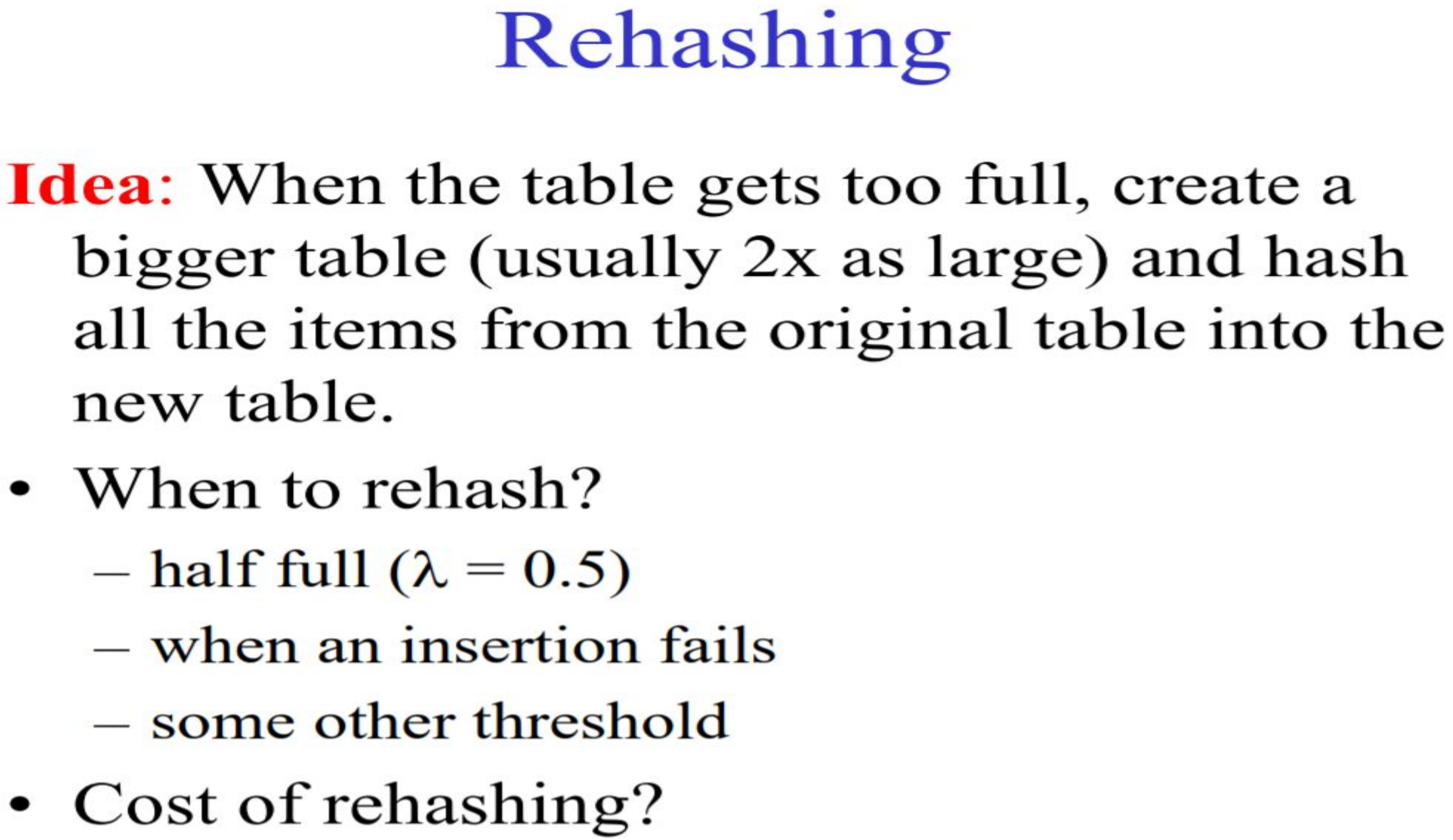
# Collision

- For example, if the item 44 had been the next item in our collection, it would have a hash value of 0 (44%11==0). Since 77 also had a hash value of 0, we would have a problem. According to the hash function, two or more items would need to be in the same slot. This is referred to as a **collision** (it may also be called a “clash”).



| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| 77 | None | None | None | 26 | 93 | 17 | None | None | 31 | 54 |

# Rehashing



## Rehashing

**Idea:** When the table gets too full, create a bigger table (usually 2x as large) and hash all the items from the original table into the new table.

- When to rehash?
  - half full ($\lambda = 0.5$)
  - when an insertion fails
  - some other threshold
- Cost of rehashing?

# Hash function

- To achieve a good hashing mechanism, It is important to have a good hash function with the following basic requirements:
- **Easy to compute:**
- It should be easy to compute and must not become an algorithm in itself.
- **Uniform distribution:**
- It should provide a uniform distribution across the hash table and should not result in clustering.
- **Less collisions:**
- Collisions occur when pairs of elements are mapped to the same hash value. These should be avoided.

# Choice of Hash Function

- Requirements
    - easy to compute
    - minimal number of collisions
- If a hashing function groups key values together, this is called clustering of the keys.
- A good hashing function distributes the key values uniformly throughout the range.

# Types of Hash Functions

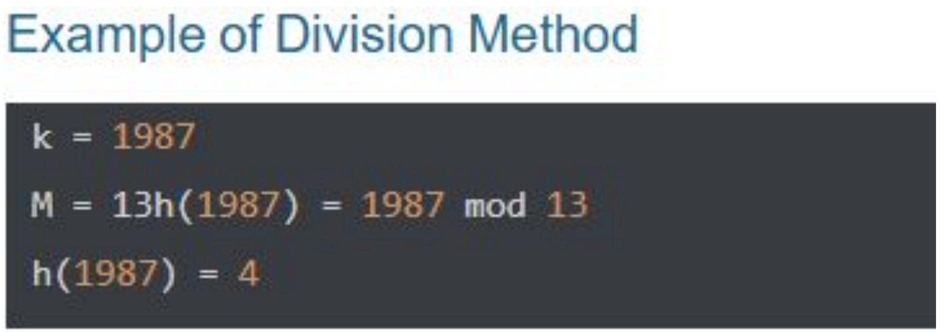
The primary types of hash functions are:

1. Division Method.
2. Mid Square Method.
3. Folding Method.
4. Multiplication Method.
5. Universal Hashing

# Types of Hash Functions

## 1. Division Method.

h(K) = k mod M
(where k = key value and M = the size of the hash table)



### Example of Division Method

```
k = 1987
M = 13h(1987) = 1987 mod 13
h(1987) = 4
```

### Advantages:

- This method is effective for all values of M.
- The division strategy only requires one operation, thus it is quite quick.

### Disadvantages:

- Since the hash table maps consecutive keys to successive hash values, this could result in poor performance.

# Types of Hash Functions

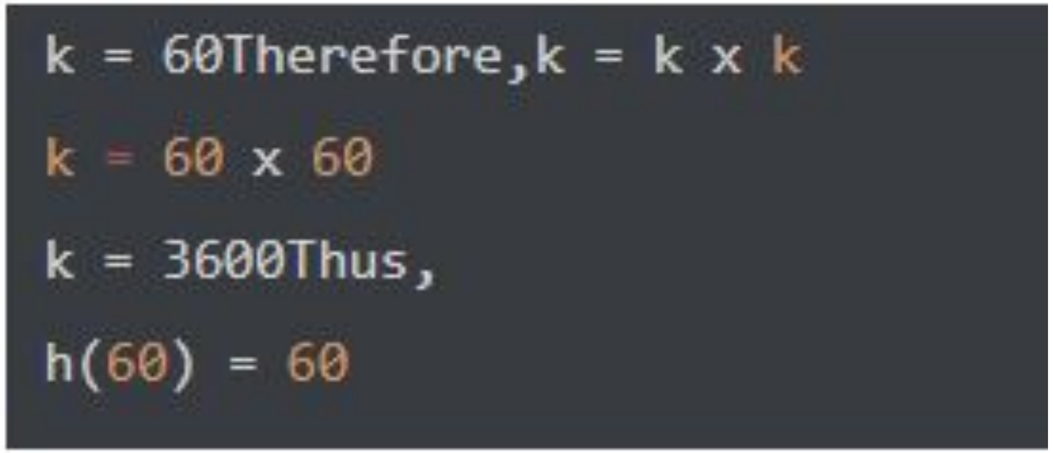
## Mid Square Method

Steps

- k*k, or square the value of k
- Using the middle r digits, calculate the hash value.

## Formula:

h(K) = h(k x k)
(where k = key value)



```
k = 60Therefore,k = k x k

k = 60 x 60

k = 3600Thus,

h(60) = 60
```

# Types of Hash Functions

## Mid Square Method

### Advantages:

- This technique works well because most or all of the digits in the key value affect the result. All of the necessary digits participate in a process that results in the middle digits of the squared result.
- The result is not dominated by the top or bottom digits of the initial key value.

### Disadvantages:

- The size of the key is one of the limitations of this system; if the key is large, its square will contain twice as many digits.
- Probability of collisions occurring repeatedly.

# Types of Hash Functions

## Folding Method

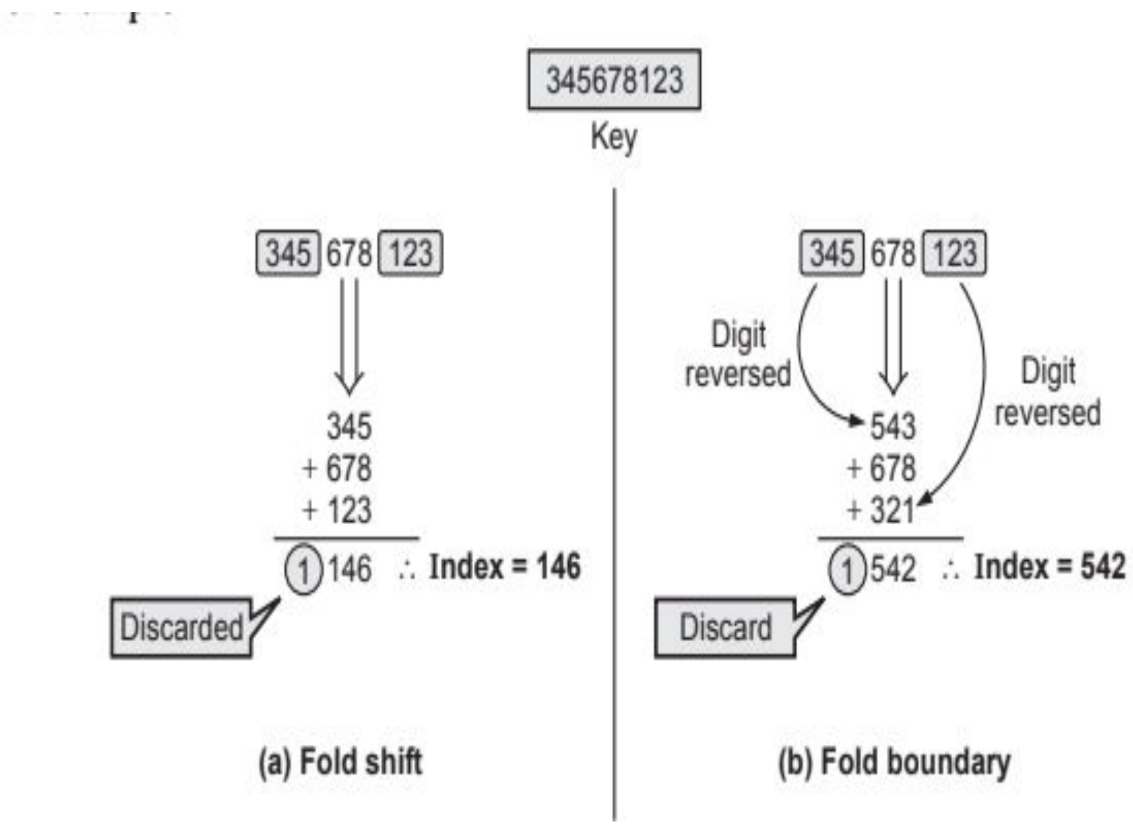
### Formula:

k = k1, k2, k3, k4, ….., kn

s = k1+ k2 + k3 + k4 +….+ kn

h(K)= s

(Where, s = addition of the parts of key k



(a) Fold shift

(b) Fold boundary

# Types of Hash Functions

Multiplication Method.

- Determine a constant value. A, where (0, A, 1)
- Add A to the key value and multiply.
- Consider kA's fractional portion.
- Multiply the outcome of the preceding step by M, the hash table's size.

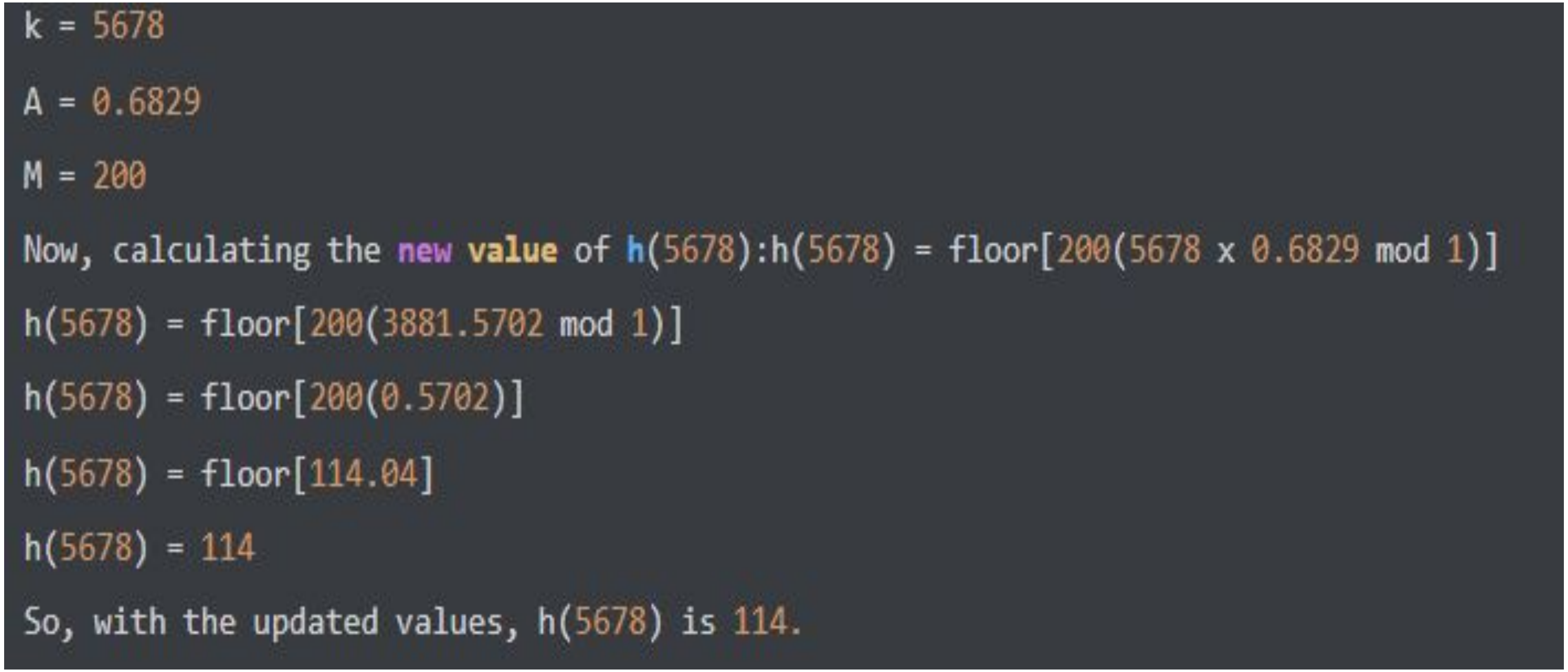
## Formula:

h(K) = floor (M (kA mod 1))

Where, M = size of the hash table, k = key value and A = constant value)

# Types of Hash Functions

## Multiplication Method- Example



```
k = 5678

A = 0.6829

M = 200

Now, calculating the new value of h(5678):h(5678) = floor[200(5678 x 0.6829 mod 1)]

h(5678) = floor[200(3881.5702 mod 1)]

h(5678) = floor[200(0.5702)]

h(5678) = floor[114.04]

h(5678) = 114

So, with the updated values, h(5678) is 114.
```

# Types of Hash Functions

## Universal Hashing

Universal hashing uses a family of hash functions to minimize the chance of collision for any given set of inputs.

$$h(k)=((a\cdot k+b)\mathrm{mod}p)\mathrm{mod}m$$

Where a and b are randomly chosen constants, p is a prime number greater than m, and k is the key.

# Types of Hash Functions

## Universal Hashing

Advantages

Reduces the probability of collisions.

Disadvantages

Requires more computation and storage.

# Some methods

- **Truncation:**
    - e.g. 123456789 map to a table of 1000 addresses by picking 3 digits of the key.
- **Folding:**
    - e.g. 123|456|789: add them and take mod.
- **Key mod N:**
    - N is the size of the table, better if it is prime.
- **Squaring:**
    - Square the key and then truncate

# Rehashing:

- **rehashing means hashing again**.
- when the load factor increases to more than its pre-defined value (default value of load factor is 0.75), the complexity increases.
- To overcome this, the size of the array is increased (doubled) and all the values are hashed again and stored in the new double sized array to maintain a low load factor and low complexity.

# Why rehashing?

- Rehashing is done because whenever key value pairs are inserted into the map, the load factor increases, which implies that the time complexity also increases as explained above. This might not give the required time complexity of O(1).
- Hence, rehash must be done, increasing the size of the bucket Array so as to reduce the load factor and the time complexity.

# How Rehashing is done?

- ***Rehashing can be done as follows:***
- For each addition of a new entry to the map, check the load factor.
- If it’s greater than its pre-defined value (or default value of 0.75 if not given), then Rehash.
- For Rehash, make a new array of double the previous size and make it the new bucketarray.
- Then traverse to each element in the old bucketArray and call the insert() for each so as to insert it into the new larger bucket array.

# Rehashing

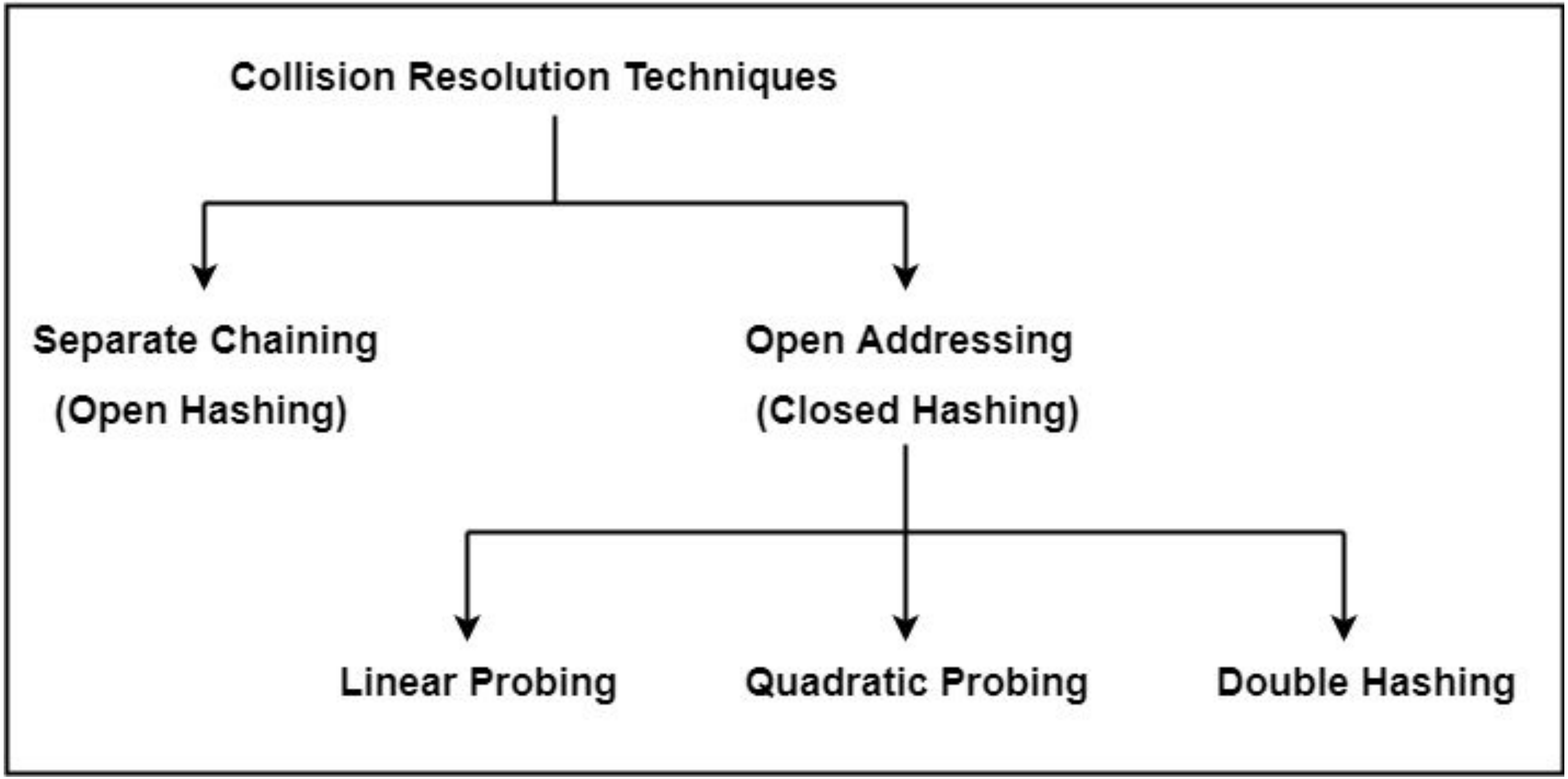
- If the table is close to full, the search time grows and may become equal to the table size.

- When the load factor exceeds a certain value (e.g. greater than 0.5) we do rehashing :

**Build a second table twice as large as the original and rehash there all the keys of the original table.**

- Rehashing is expensive operation, with running time O(N)
- However, once done, the new hash table will have good performance.

# Types of Collision Resolution Techniques-



Collision Resolution Techniques

Separate Chaining (Open Hashing)

Open Addressing (Closed Hashing)

Linear Probing

Quadratic Probing

Double Hashing

# Contrasting Closed Vs. Open Hashing

| | Closed hashing (open addressing) | Open hashing: separate chaining) |
|---|---|---|
| **Description** | •Resolve collisions by finding another place in the hash table | •Resolve collisions by inserting additional elements at the same location in the hash table |
| **Strengths** | •May be faster in practice because the table doesn't change in size | •May require less memory (smaller hash table)<br>•Fewer compares |
| **Weaknesses** | •Requires a larger hash table (percentage utilization of the table should be lower than with open hashing) | •May be slower in practice because of the dynamic memory allocations<br>•More complex: needs another data structure |

James Tam

# Collision Resolution

## Collision

- It is obvious that no matter what function is used, the possibility exists that the use of the function will produce an index which is a duplicate of an index which already exists. This is a Collision.

  Collision resolution strategy:

  - Open addressing: store the key/entry in a different position

    - Chaining: chain together several keys/entries in each position

# Collision Resolution – Open Addressing

- Resolving collisions by open addressing is resolving the problem by taking the next open space as determined by rehashing the key according to some algorithm.

- Two main open addressing collision resolution techniques:
  - - Linear probing: increase by 1 each time [mod table size!]
  - - Quadratic probing: to the original position, add 1, 4, 9, 16,…

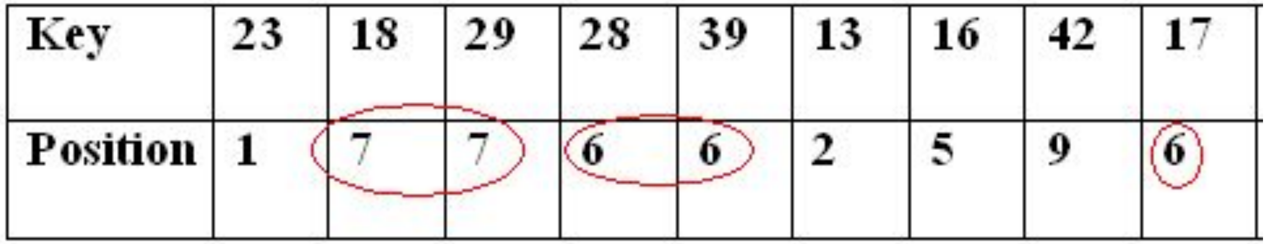
**Probing**
If the table position given by the hashed key is already occupied, increase the position by some amount, until an empty position is found

# Collision Resolution

## Collision - Example

- - Hash table size 11
- - Hash function: key mod hash size

So, the new positions in the hash table are:



| Key | 23 | 18 | 29 | 28 | 39 | 13 | 16 | 42 | 17 |
|---|---|---|---|---|---|---|---|---|---|
| Position | 1 | 7 | 7 | 6 | 6 | 2 | 5 | 9 | 6 |

Some collisions occur with this hash function.

# Collision Resolution

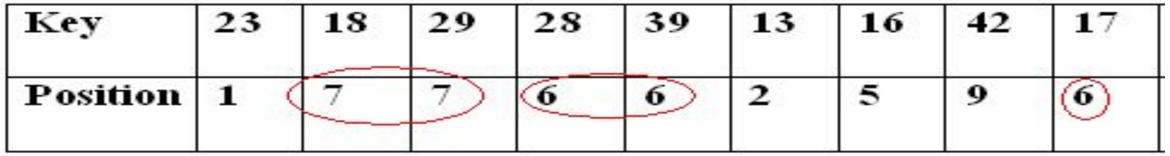
## Collision Resolution – Open Addressing

### Linear Probing

new position = (current position + 1) MOD hash size

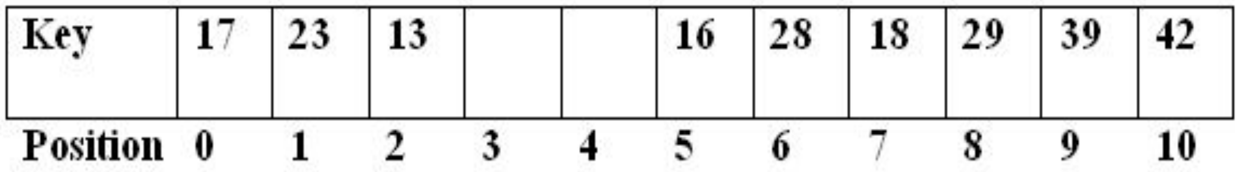
Example –

Before linear probing:



| Key | 23 | 18 | 29 | 28 | 39 | 13 | 16 | 42 | 17 |
|---|---|---|---|---|---|---|---|---|---|
| Position | 1 | 7 | 7 | 6 | 6 | 2 | 5 | 9 | 6 |

After linear probing:



| Key | 17 | 23 | 13 | | | 16 | 28 | 18 | 29 | 39 | 42 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Position | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

Problem – Clustering occurs, that is, the used spaces tend to appear in groups which tends to grow and thus increase the search time to reach an open space.

Example:

Insert items with keys:

**89, 18, 49, 58, 9**

into an empty hash table.

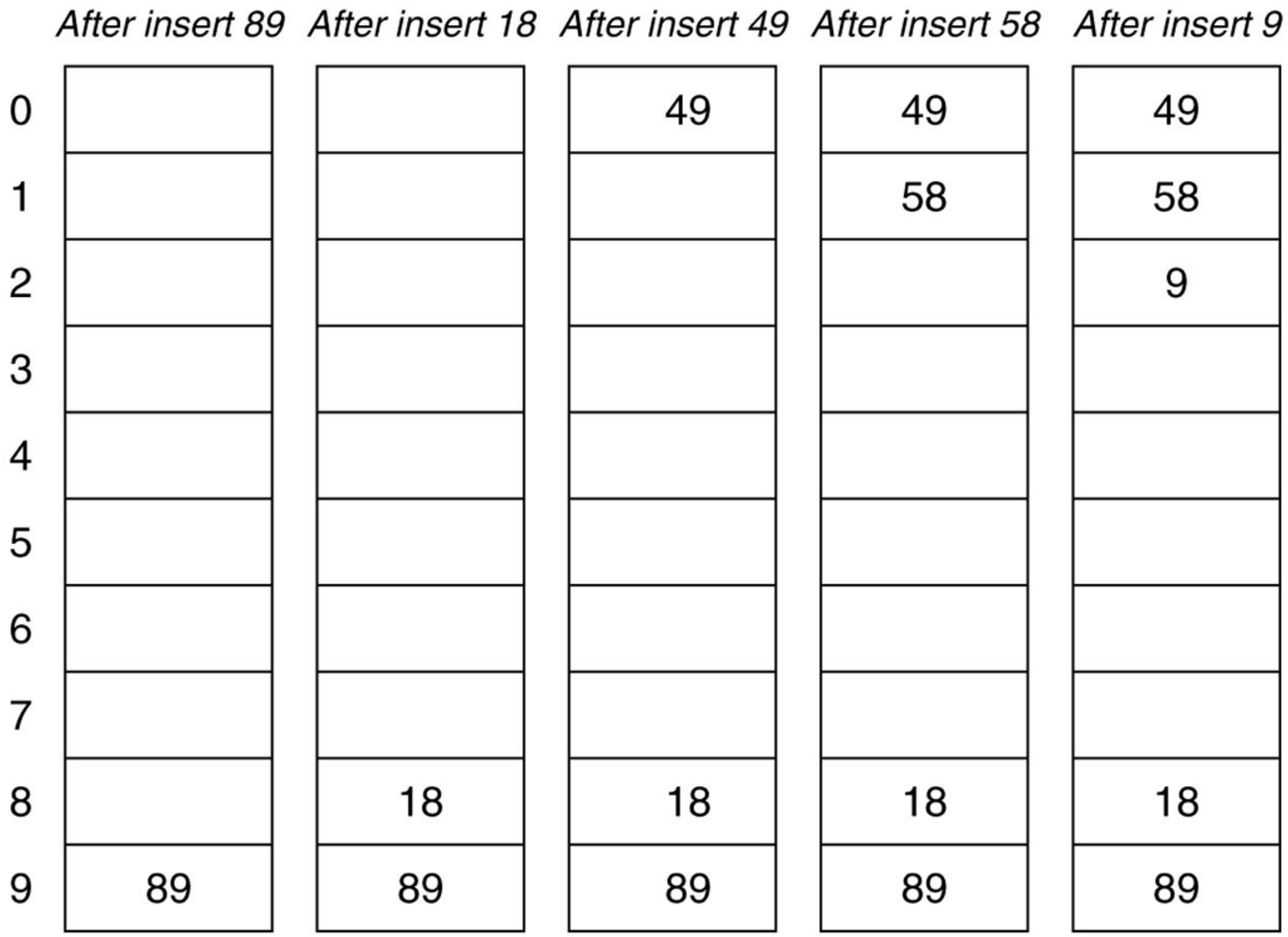
**Table size is 10**

Hash function is :

**hash(x) = x mod 10**

## Linear probing hash table after each insertion

```
hash ( 89, 10 ) = 9
hash ( 18, 10 ) = 8
hash ( 49, 10 ) = 9
hash ( 58, 10 ) = 8
hash (  9, 10 ) = 9
```

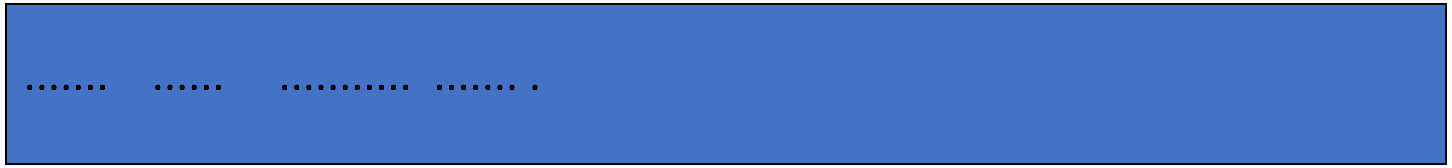


|   | *After insert 89* | *After insert 18* | *After insert 49* | *After insert 58* | *After insert 9* |
|---|---|---|---|---|---|
| 0 |  |  | 49 | 49 | 49 |
| 1 |  |  |  | 58 | 58 |
| 2 |  |  |  |  | 9 |
| 3 |  |  |  |  |  |
| 4 |  |  |  |  |  |
| 5 |  |  |  |  |  |
| 6 |  |  |  |  |  |
| 7 |  |  |  |  |  |
| 8 |  | 18 | 18 | 18 | 18 |
| 9 | 89 | 89 | 89 | 89 | 89 |

# Linear Probing

- Eliminates need for separate data structures (chains), and the cost of constructing nodes.

- Leads to problem of clustering. Elements tend to cluster in dense intervals in the array.



- Search efficiency problem remains.
- Deletion becomes trickier….

- **Collision Resolution – Open Addressing**

□ In order to try to avoid clustering, a method which does not look for the first open space must be used.

□ Solution to avoid clustering:

- Quadratic Probing

# Collision Resolution

## Collision Resolution – Open Addressing

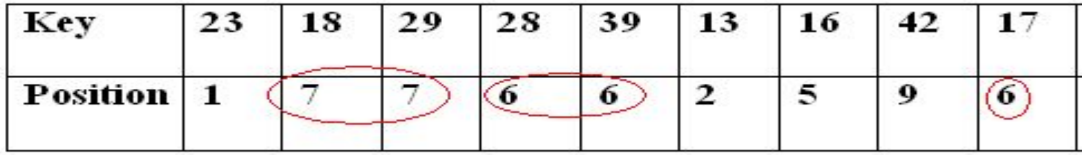
### Quadratic Probing

new position = (collision position + $j^2$) MOD hash size

{ j = 1, 2, 3, 4, ……}

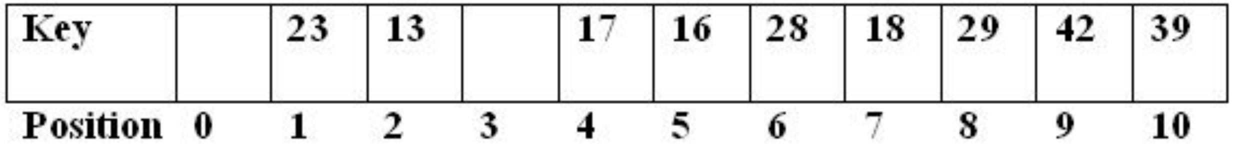
Example –

Before quadratic probing:



| Key | 23 | 18 | 29 | 28 | 39 | 13 | 16 | 42 | 17 |
|---|---|---|---|---|---|---|---|---|---|
| Position | 1 | 7 | 7 | 6 | 6 | 2 | 5 | 9 | 6 |

After quadratic probing:



| Key |  | 23 | 13 |  | 17 | 16 | 28 | 18 | 29 | 42 | 39 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Position | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

Problem – Overflow may occurs when there is still space in the hash table.

Example:

Insert items with keys:

**89, 18, 49, 58, 9**

into an empty hash table.

**Table size is 10**

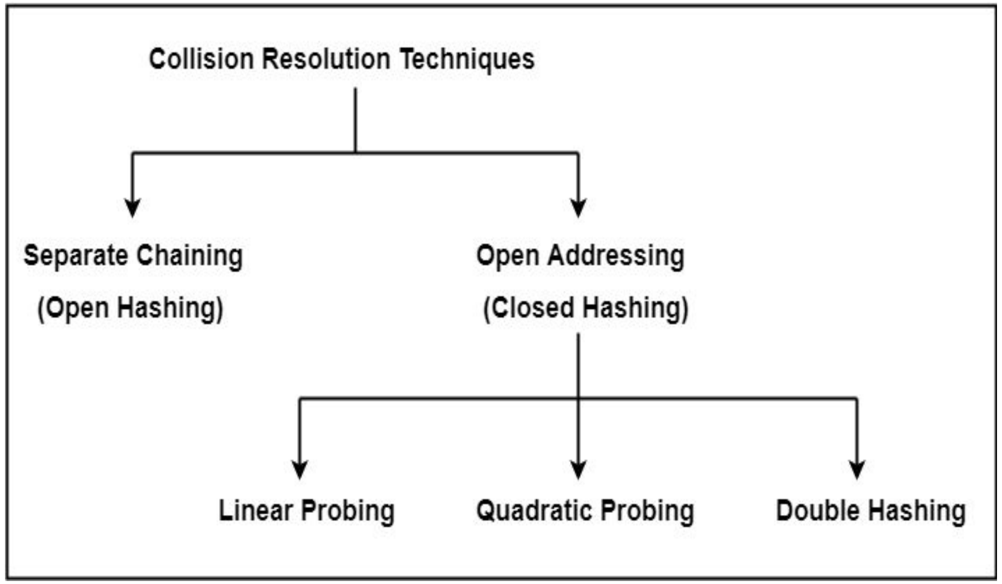
## A quadratic probing hash table after each insertion

```
hash ( 89, 10 ) = 9
hash ( 18, 10 ) = 8
hash ( 49, 10 ) = 9
hash ( 58, 10 ) = 8
hash (  9, 10 ) = 9
```

| | *After insert 89* | *After insert 18* | *After insert 49* | *After insert 58* | *After insert 9* |
|---|---|---|---|---|---|
| 0 | | | 49 | 49 | 49 |
| 1 | | | | | |
| 2 | | | | 58 | 58 |
| 3 | | | | | 9 |
| 4 | | | | | |
| 5 | | | | | |
| 6 | | | | | |
| 7 | | | | | |
| 8 | | 18 | 18 | 18 | 18 |
| 9 | 89 | 89 | 89 | 89 | 89 |

# Double Hashing

- **Double hashing** is a collision resolving technique in **Open Addressed** Hash tables.
- Double hashing uses the idea of applying a second hash function to key when a collision occurs.
- First hash function is typically **hash1(key) = key % TABLE_SIZE**
- A popular second hash function is : **hash2(key) = PRIME – (key % PRIME)** where PRIME is a prime smaller than the TABLE_SIZE.
- A good second Hash function is:
- It must never evaluate to zero
- Must make sure that all cells can be probed



## Collision Resolution Techniques

- There are two broad ways of collision resolution:

1. Separate Chaining: An array of linked list implementation.

2. Open Addressing: Array-based implementation.

   (i) Linear probing (linear search)
   (ii) Quadratic probing (nonlinear search)
   (iii) Double hashing (uses two hash functions)

# **Separate Chaining- (**Open Hashing )

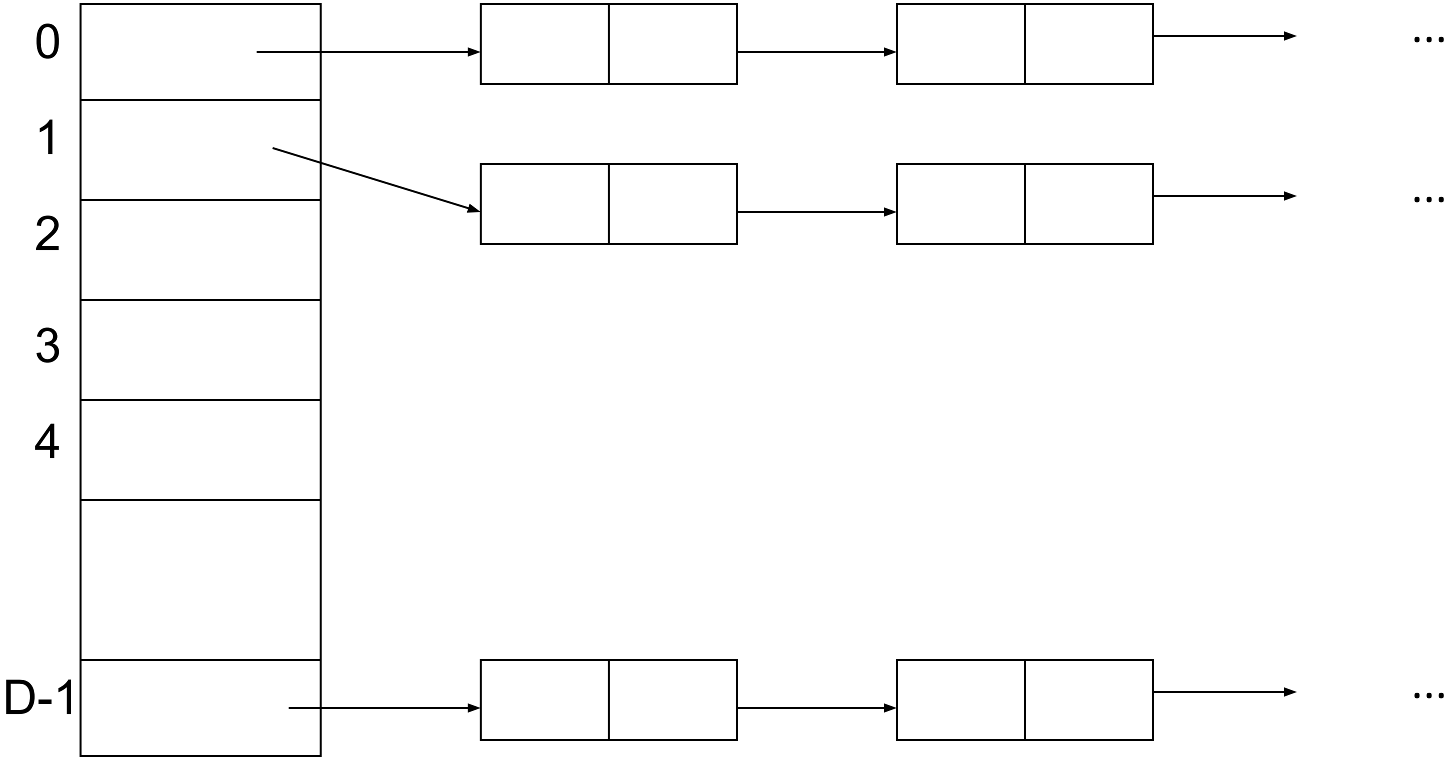
This technique creates separate linked lists for the slots which appear like chains, therefore this technique is called as **“Separate Chaining”**.

# **Separate Chaining- (**Open Hashing )

- This technique handles the collision by making those slots of the hash table for which collision occurs pointing to a linked list of records.
- when a new key is being inserted and
- if the slot to which it maps is already occupied, then **??**
- linked list is created for that slot and the new key is inserted in the list.

# Open Hashing Data Organization

## **Separate Chaining-**



Each bucket in the hash table is the head of a linked list

All elements that hash to a particular bucket are placed on that bucket’s linked list

**Example:** Load the keys **23, 13, 21, 14, 7, 8, and 15** , in this order, in a hash table of size **7** using separate chaining with the hash function: **h(key) = key % 7**

h(23) = 23 % 7 = 2

h(13) = 13 % 7 = 6

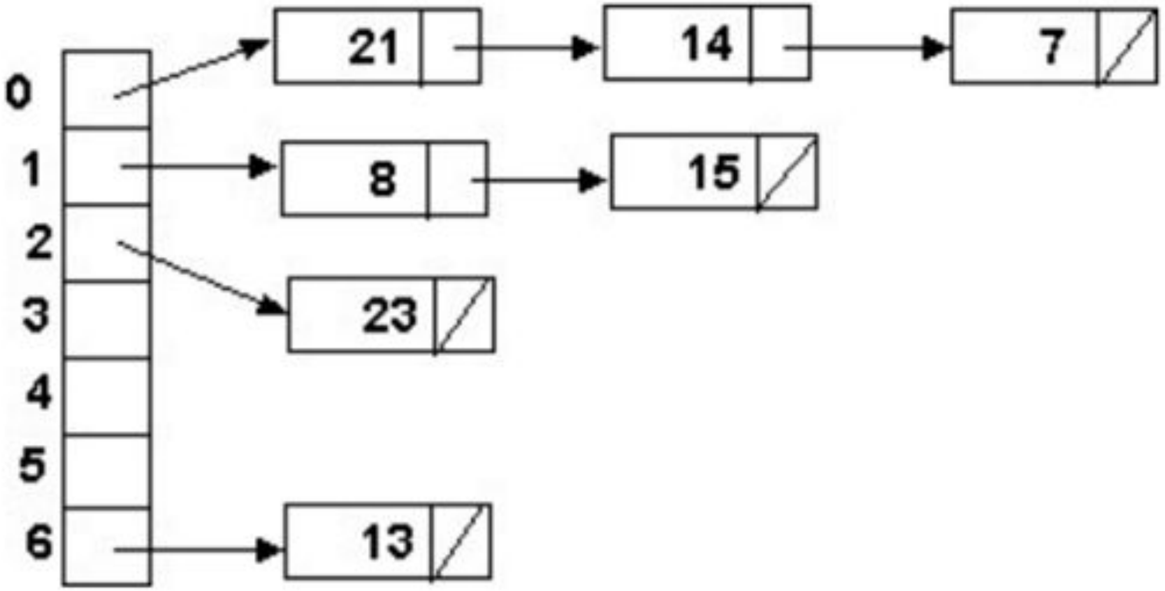
h(21) = 21 % 7 = 0

h(14) = 14 % 7 = 0 collision

h(7) = 7 % 7 = 0 collision

h(8) = 8 % 7 = 1

h(15) = 15 % 7 = 1 collision



# Collision Resolution – Chaining

□ Each table position is a linked list

□ Add the keys and entries anywhere in the list (front easiest)

Advantages over open addressing:

- Simpler insertion and removal
- Array size is not a limitation (but should still minimize collisions: make table size roughly equal to expected number of keys and entries)

Disadvantage

- Memory overhead is large if entries are small

# Open hashing or separate chaining

- Open hashing  is a collision avoidance method which uses array of linked list to resolve the collision.
- It is also known as the separate chaining method (each linked list is considered as a chain).

# **Algorithm** | *Insert data into the separate chain*

- 1. Declare an array of a linked list with the hash table size.
- 2. Initialize an array of a linked list to NULL.
- 3. Find hash key.
- 4. If chain[key] == NULL
-     Make chain[key] points to the key node.
- 5. Otherwise(collision),
-     Insert the key node at the end of the chain[key].

# Separate chaining implementation in c

```
#include<stdio.h>
#include<stdlib.h>

#define size 7

struct node
{
   int data;
    struct node *next;
};

struct node *chain[size];
```

# Initialization

```
void init()
{
    int i;
    for(i = 0; i < size; i++)
     chain[i] = NULL;
}
```

```
void insert(int value)
{
    //create a newnode with value
    struct node *newNode = malloc(sizeof(struct node));
    newNode->data = value;
    newNode->next = NULL;
    //calculate hash key
    int key = value % size;
    //check if chain[key] is empty
    if(chain[key] == NULL)
        chain[key] = newNode;
    //collision
    else
    {
   //add the node at the end of chain[key].
        struct node *temp = chain[key];
        while(temp->next)
        {
            temp = temp->next;
        }
        temp->next = newNode;
    }
}
```

```
void print()
{
    int i;

    for(i = 0; i < size; i++)
    {
        struct node *temp = chain[i];
        printf("chain[%d]-->",i);
        while(temp)
        {
            printf("%d -->",temp->data);
            temp = temp->next;
        }
        printf("NULL\n");
    }
}
```

```
int main()
{
    //init array of list to NULL
    init();

    insert(7);
    insert(0);
    insert(3);
    insert(10);
    insert(4);
    insert(5);

    print();
    return 0;
}
```

## Output

```
chain[0]-->7 -->0 -->NULL

chain[1]-->NULL

chain[2]-->NULL

chain[3]-->3 -->10 -->NULL

chain[4]-->4 -->NULL

chain[5]-->5 -->NULL

chain[6]-->NULL
```

# What is static hashing and dynamic hashing?

- The problem with **static hashing** is that it does not expand or shrink **dynamically** as the size of the database grows or shrinks.
- **Dynamic hashing** provides a mechanism in which data buckets are added and removed **dynamically** and on-demand. **Dynamic hashing** is also known as extended **hashing**.

# What is extendible hashing in data structure?

- **Extendible hashing** is a type of **hash** system which treats a **hash** as a bit string, and uses a trie for bucket lookup

- In computer science, a **trie**, also called digital tree, radix tree or prefix tree, is a kind of search tree—an ordered tree data structure used to store a dynamic set or associative array where the keys are usually strings.

# Directory of Pointers

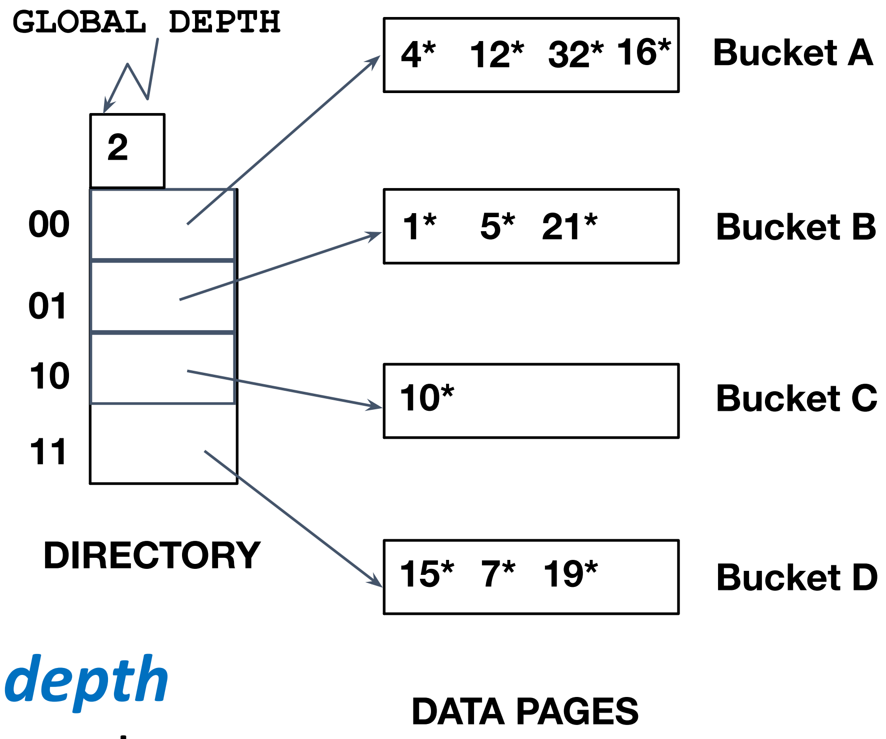
- How else (*as opposed to overflow pages*) can we add a data record to a full bucket in a *static* hash file?
  - Reorganize the table (e.g., by doubling the number of buckets and redistributing the entries across the new set of buckets)
  - But, reading and writing all pages is expensive!

- In contrast, we can use a directory of pointers to buckets
  - Buckets number can be doubled by doubling just the directory and *splitting "only" the bucket that overflowed*
  - The *trick* lies on how the hash function can be adjusted!

# Extendible Hashing

- Extendible Hashing uses a directory of pointers to buckets

- The result of applying a hash function ***h*** is treated as a *binary number* and the last ***d*** bits are interpreted as an offset into the directory

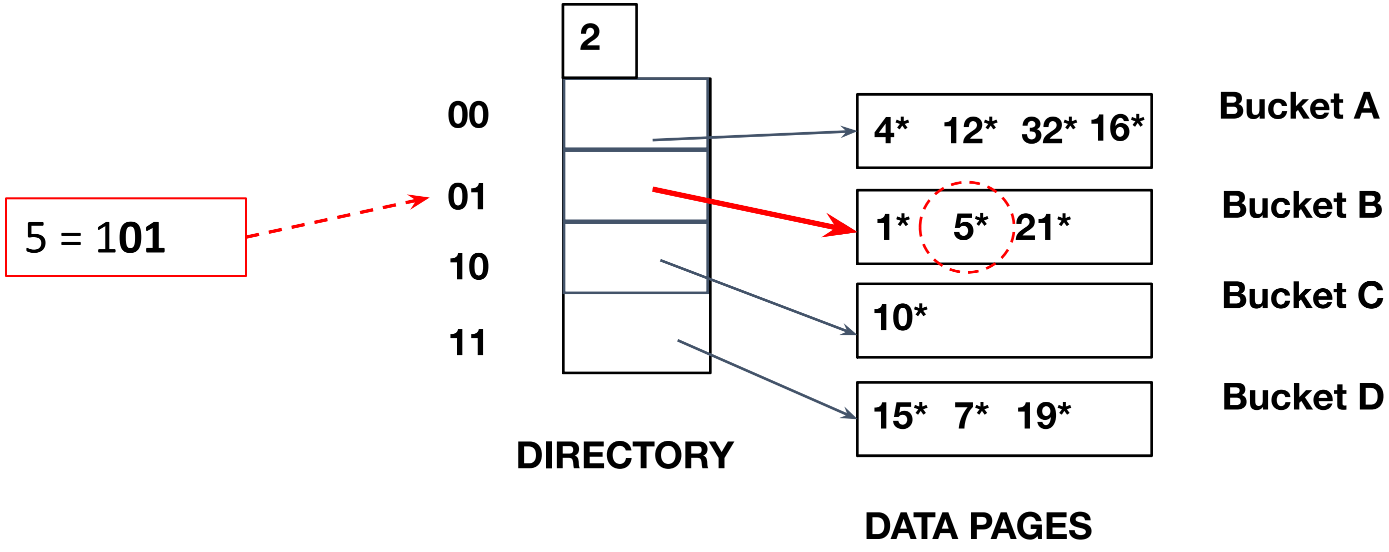


- ***d*** is referred to as the ***global depth*** of the hash file and is kept as part of the header of the file

# Extendible Hashing: Searching for Entries

- To search for a data entry, apply a hash function ***h*** to the key and take the last ***d*** bits of its binary representation to get the bucket number

- Example: search for 5*



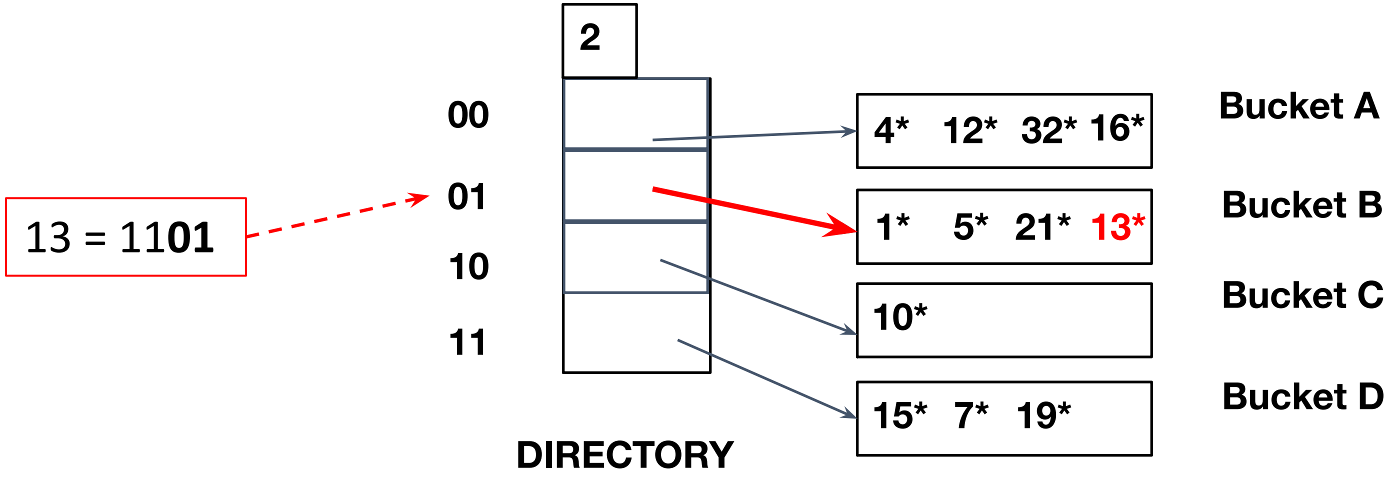
# Extendible Hashing: Inserting Entries

- An entry can be inserted as follows:
    - Find the appropriate bucket (*as in search*)
    - Split the bucket *if full* and *redistribute* contents (including the new entry to be inserted) across the old bucket and its ***"split image"***
    - Double the directory *if necessary*
    - Insert the given entry

# Extendible Hashing: Inserting Entries

- Find the appropriate bucket (as in search), split the bucket if full, double the directory if necessary and insert the given entry

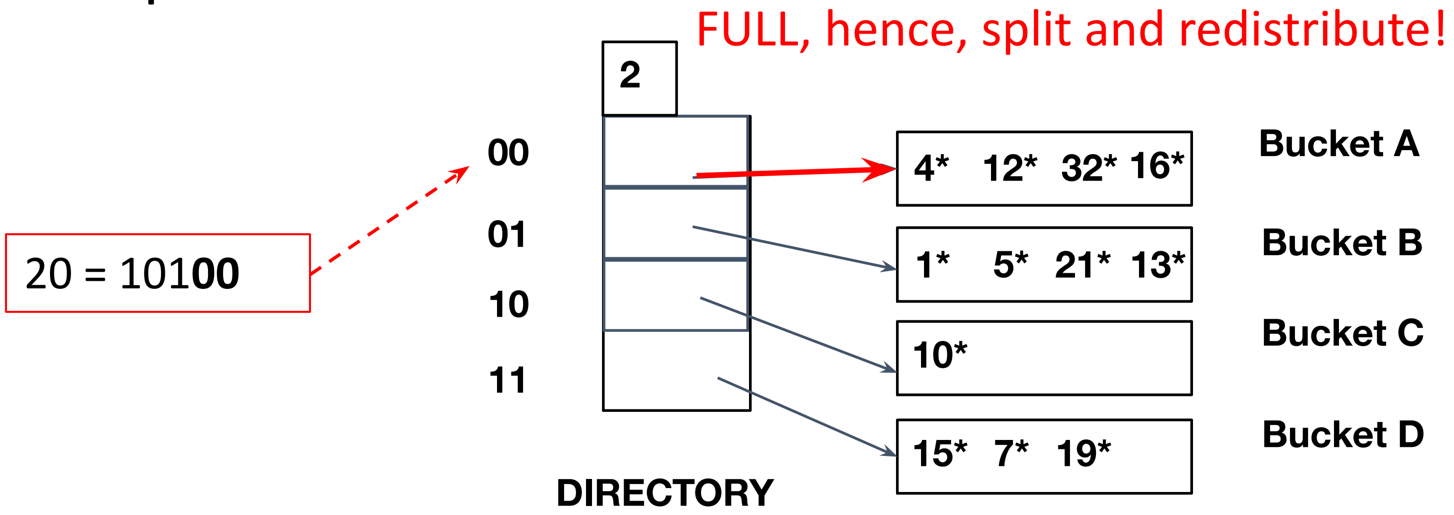
- Example: insert 13*



# Extendible Hashing: Inserting Entries

- Find the appropriate bucket (as in search), split the bucket if full, double the directory if necessary and insert the given entry

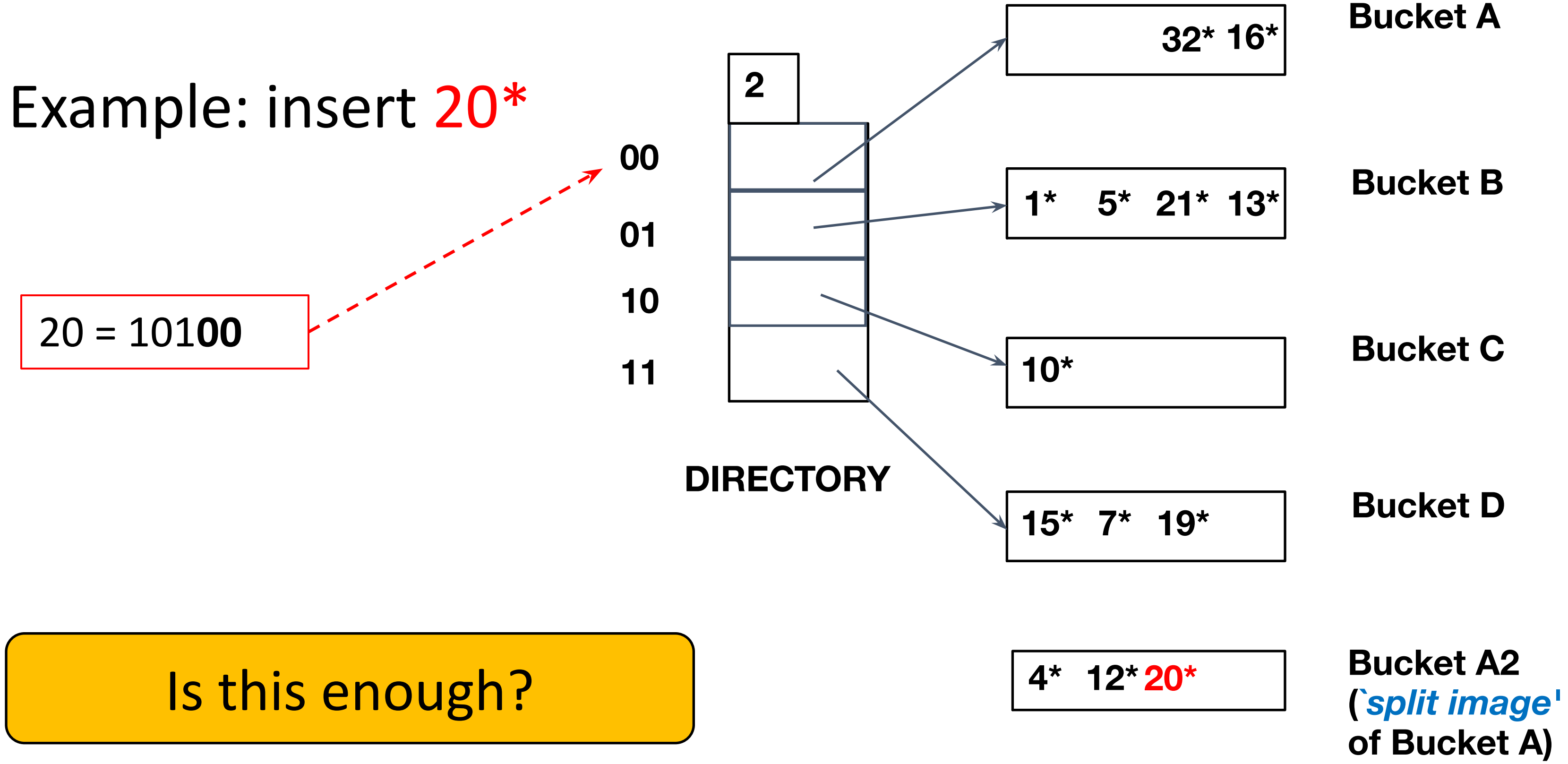
- Example: insert 20*



# Extendible Hashing: Inserting Entries

- Find the appropriate bucket (as in search), split the bucket if full, double the directory if necessary and insert the given entry

- Example: insert 20*

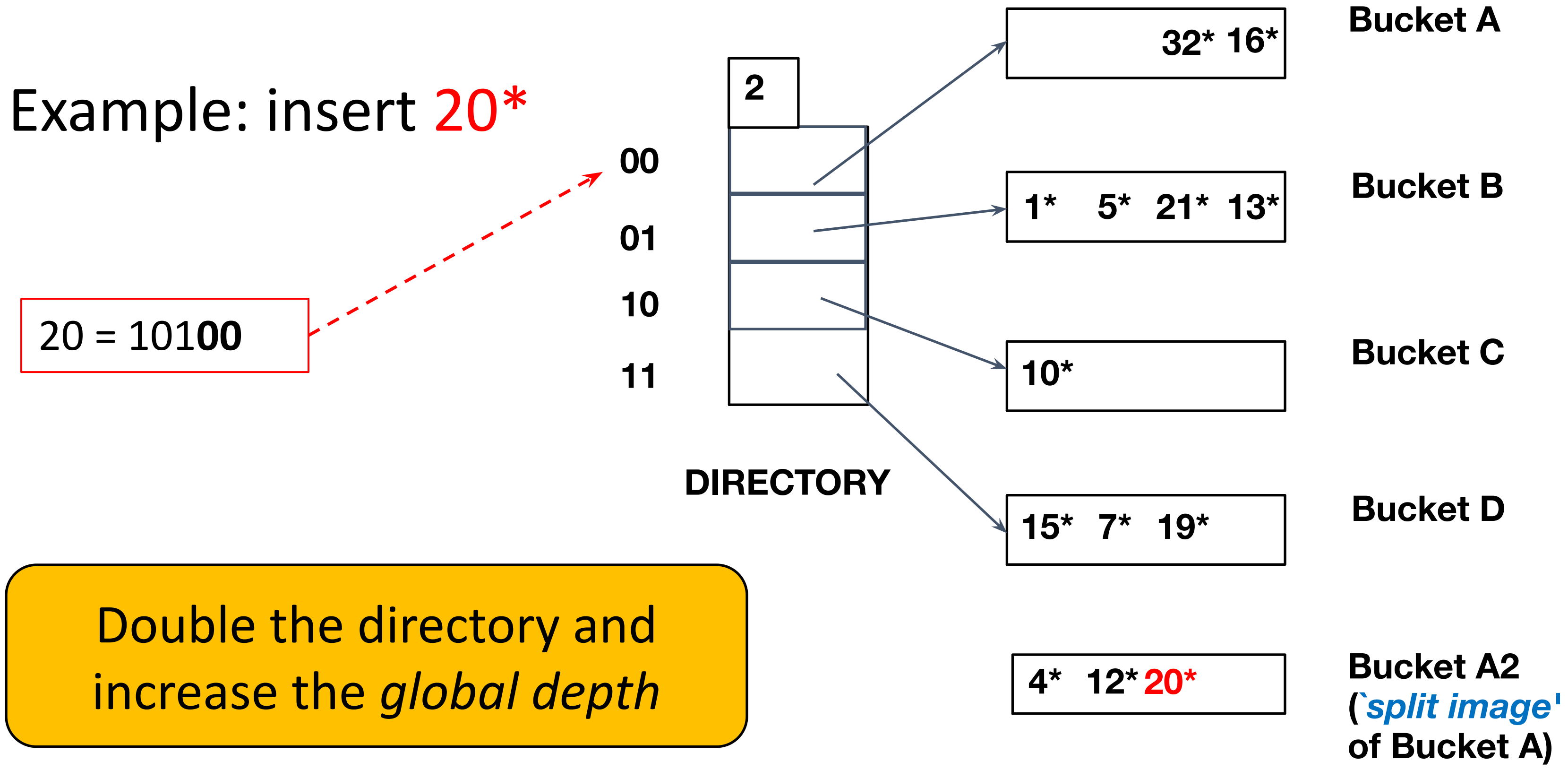


Is this enough?

# Extendible Hashing: Inserting Entries

- Find the appropriate bucket (as in search), split the bucket if full, double the directory if necessary and insert the given entry

- Example: insert 20*

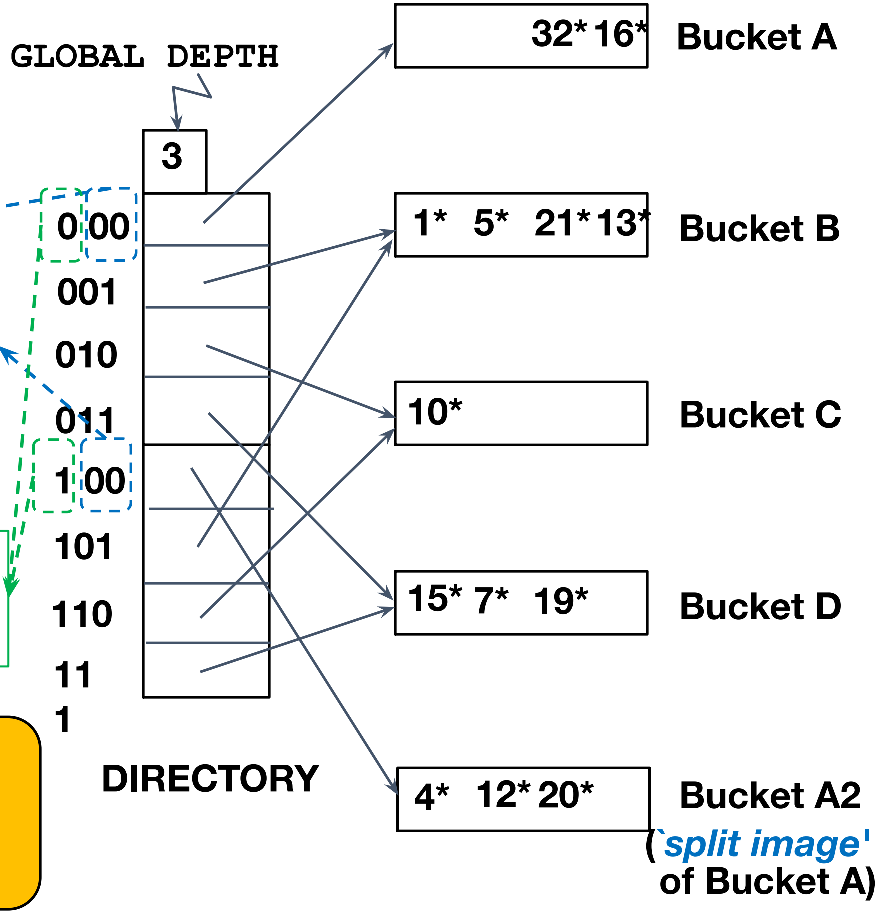


Double the directory and increase the *global depth*

# Extendible Hashing: Inserting Entries

- Find the appropriate bucket (as in search), split the bucket if full, double the directory if necessary and insert the given entry

- Example: insert 20*



These two bits indicate a data entry that belongs to one of these two buckets

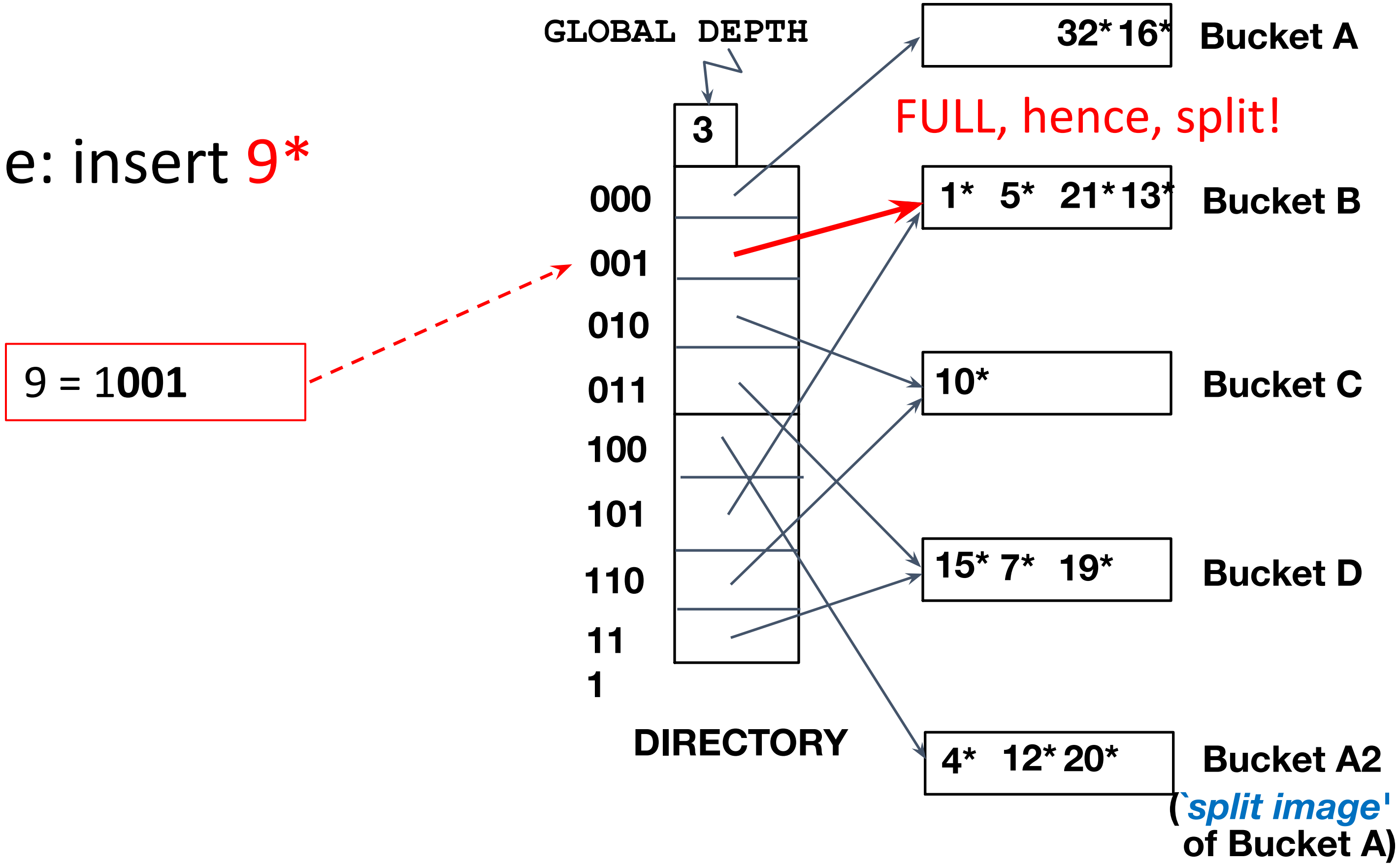
The third bit distinguishes between these two buckets!

But, is it necessary always to double the directory?

# Extendible Hashing: Inserting Entries

- Find the appropriate bucket (as in search), split the bucket if full, double the directory if necessary and insert the given entry

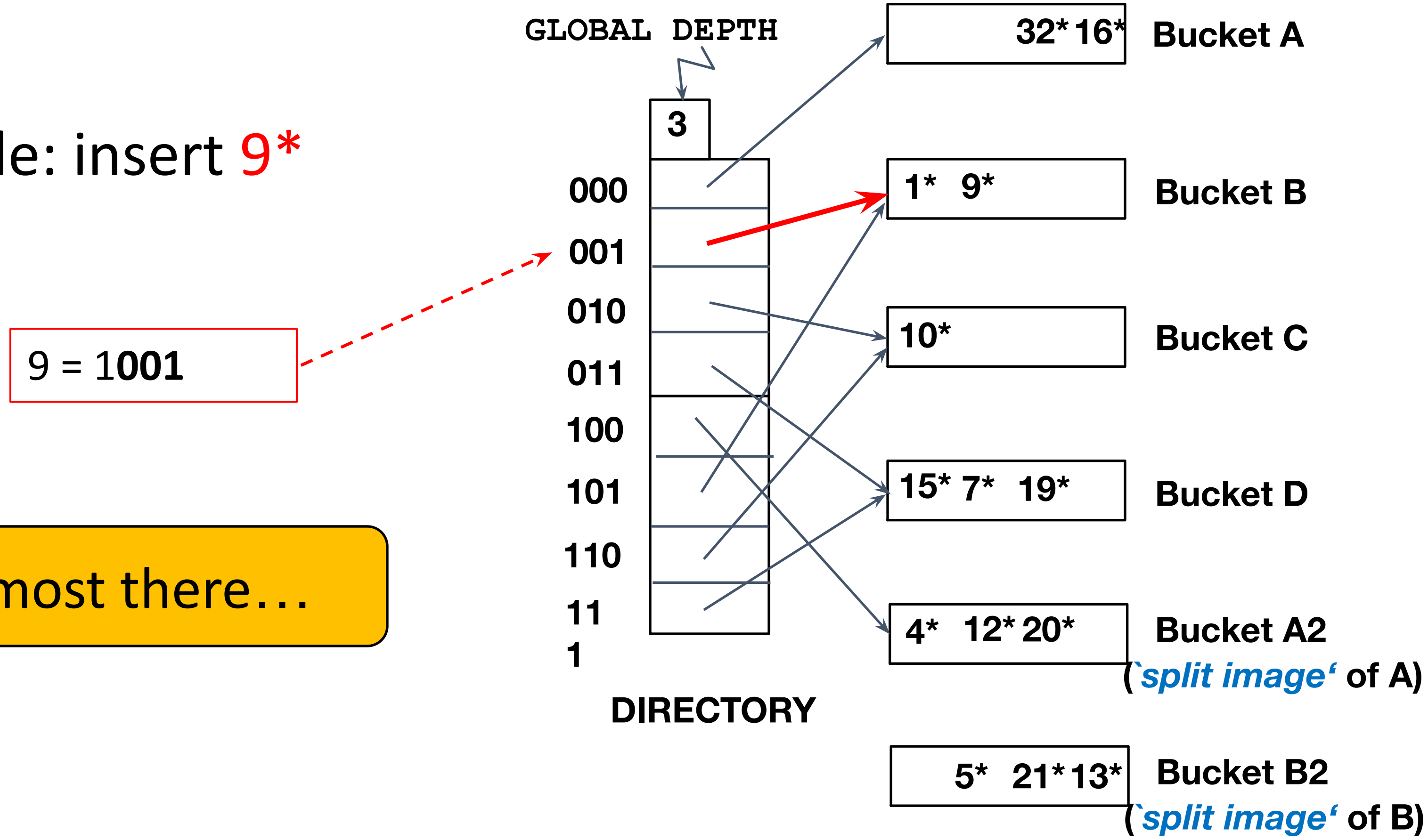
- Example: insert 9*



# Extendible Hashing: Inserting Entries

- Find the appropriate bucket (as in search), split the bucket if full, double the directory if necessary and insert the given entry

- Example: insert 9*



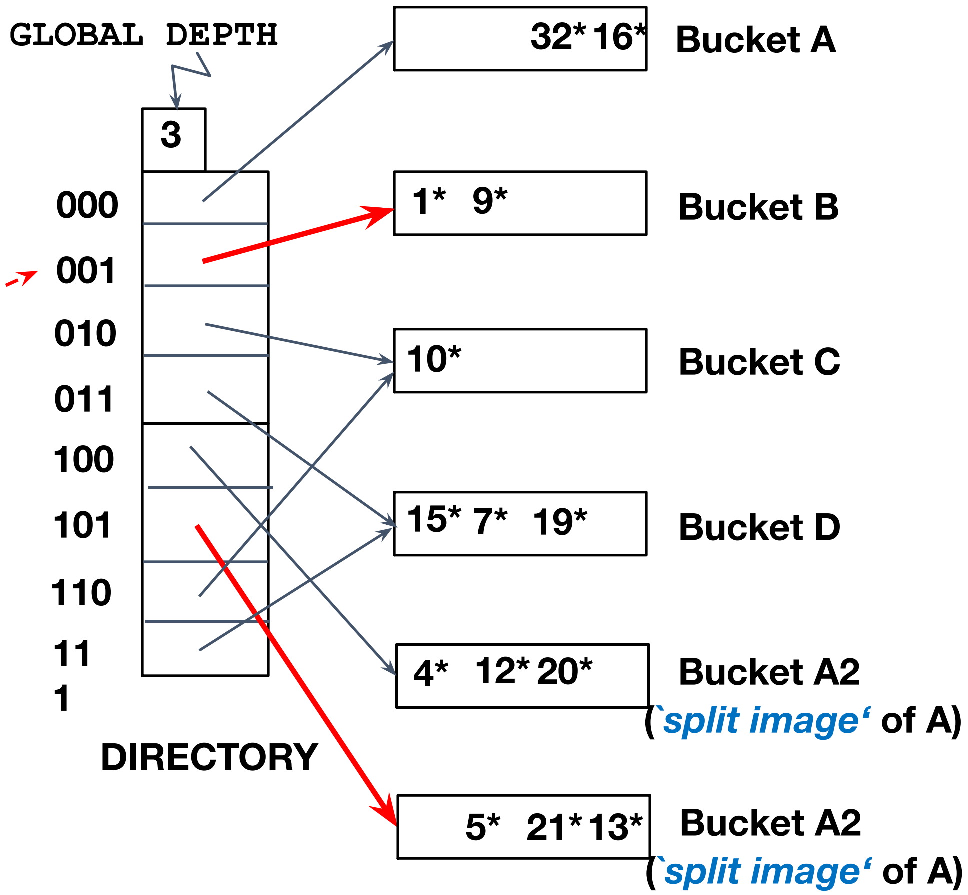
Almost there…

# Extendible Hashing: Inserting Entries

- Find the appropriate bucket (as in search), split the bucket if full, double the directory if necessary and insert the given entry

- Example: insert 9*

9 = 1001



There was no need to double the directory!

When NOT to double the directory?

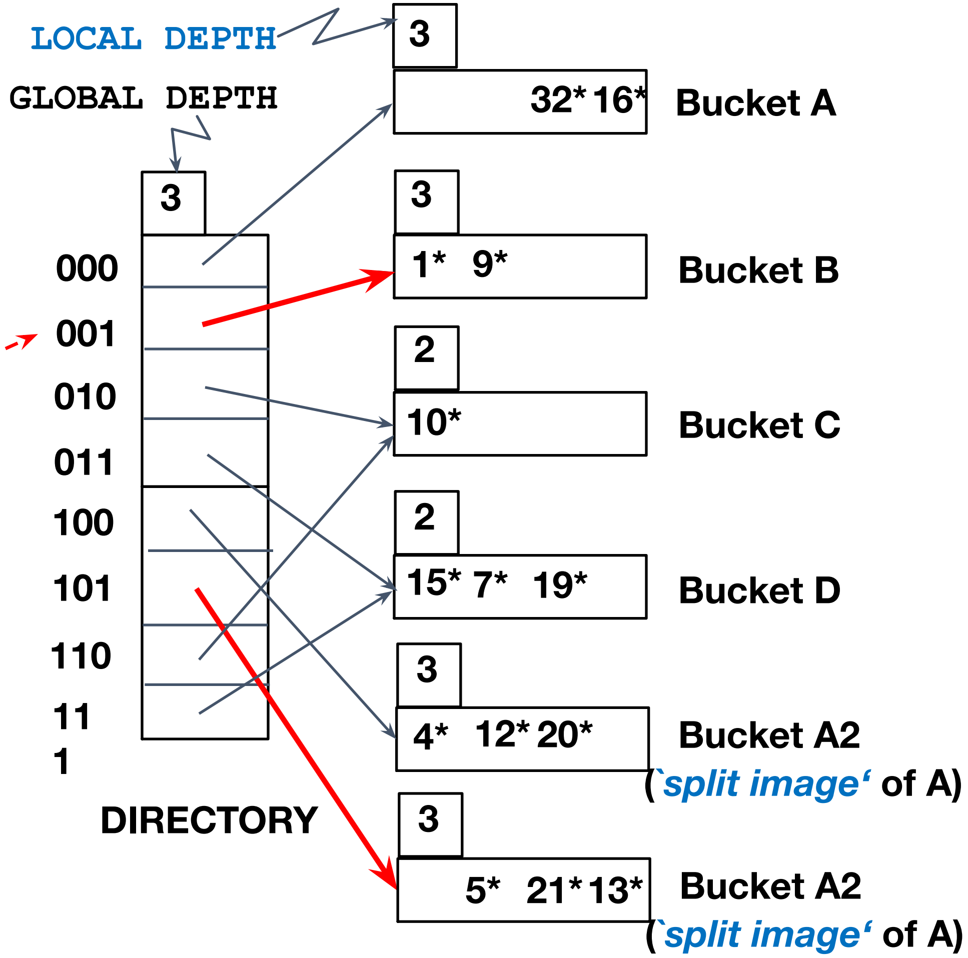
# Extendible Hashing: Inserting Entries

- Find the appropriate bucket (as in search), split the bucket if full, double the directory if necessary and insert the given entry

- Example: insert 9*

9 = 1001

If a bucket whose **local depth equals to the global depth** is split, the directory *must* be doubled



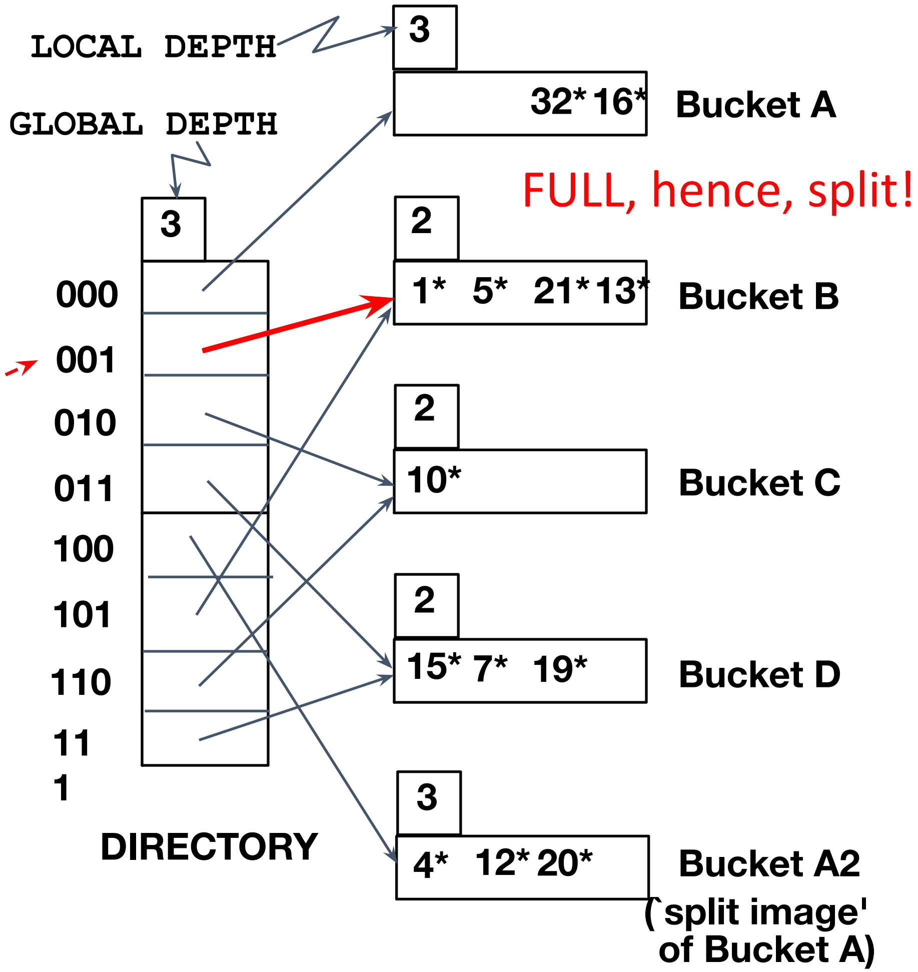
# Extendible Hashing: Inserting Entries

- Example: insert 9*

Repeat…

9 = 1001

Because the local depth (i.e., 2) is *less than* the global depth (i.e., 3), NO need to double the directory

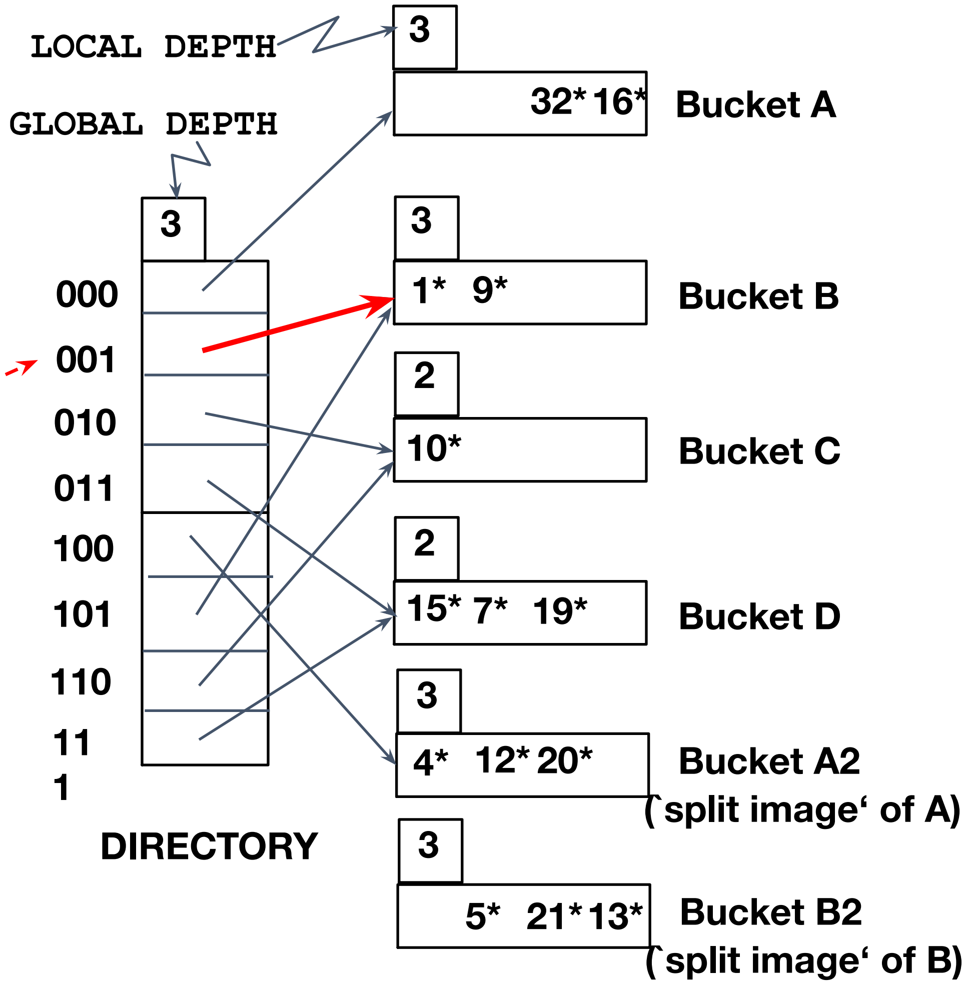


# Extendible Hashing: Inserting Entries

- Example: insert 9*

Repeat…

9 = 1001



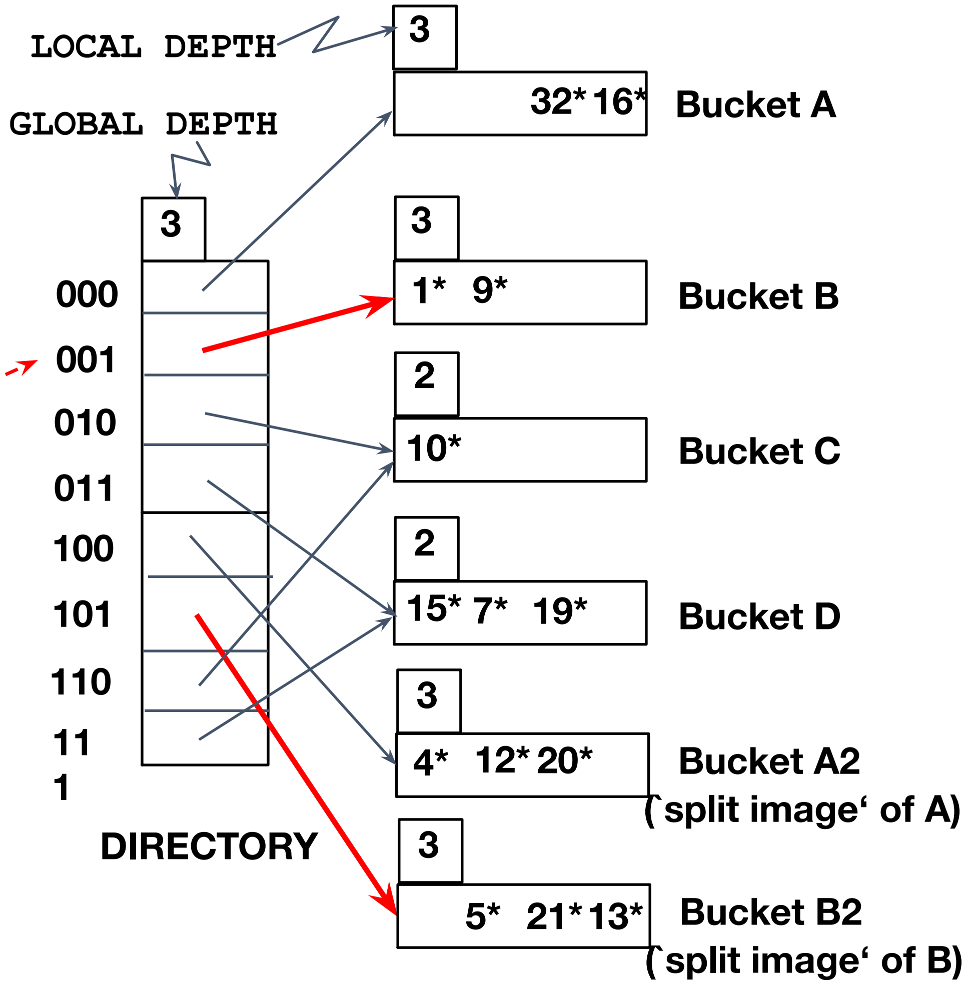
# Extendible Hashing: Inserting Entries

- Example: insert 9*

Repeat…

9 = 1001

FINAL STATE!

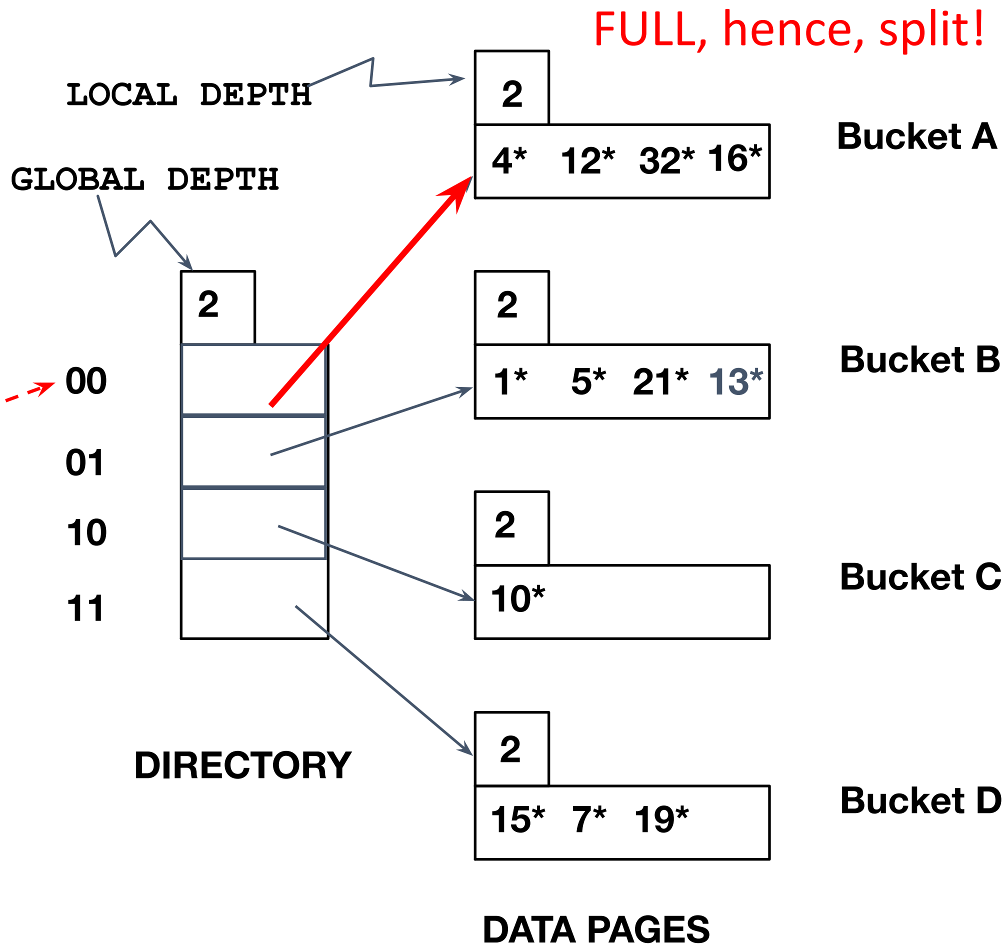


# Extendible Hashing: Inserting Entries

- Example: insert 20*

Repeat…

20 = 101**00**



Because the local depth and the global depth are both 2, we *should* double the directory!

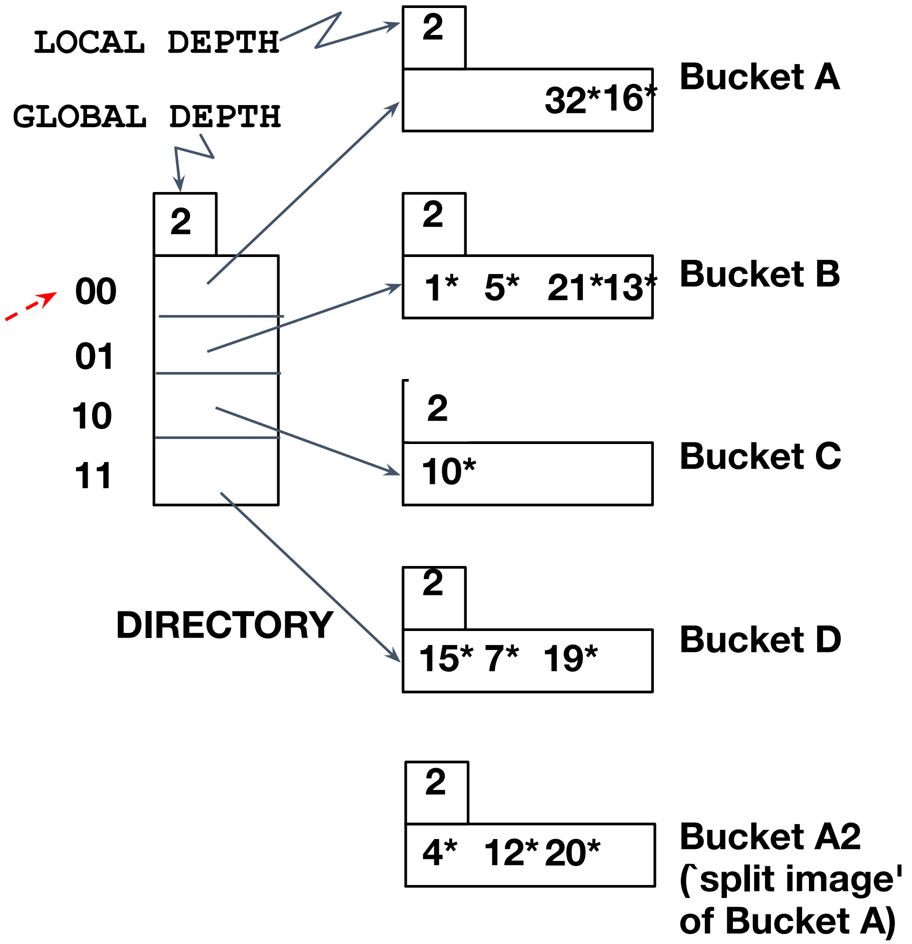
# Extendible Hashing: Inserting Entries

- Example: insert 20*

Repeat…

20 = 10100

Is this enough?

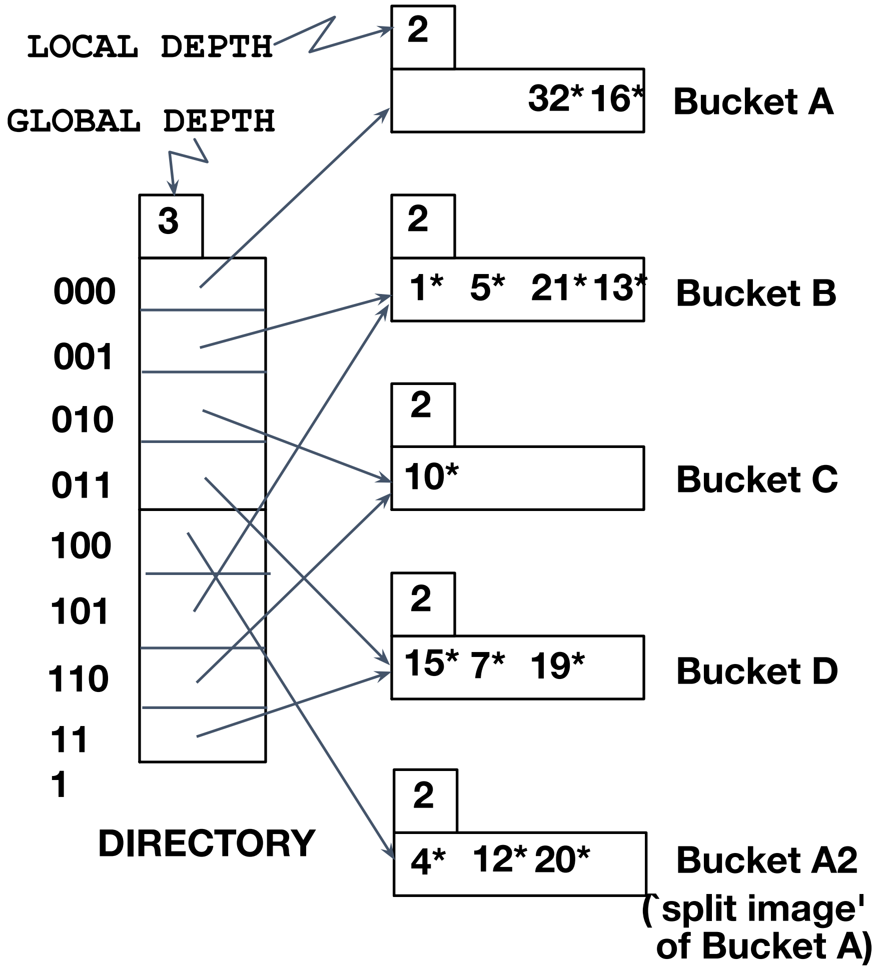


# Extendible Hashing: Inserting Entries

- Example: insert 20*

Repeat…

Is this enough?

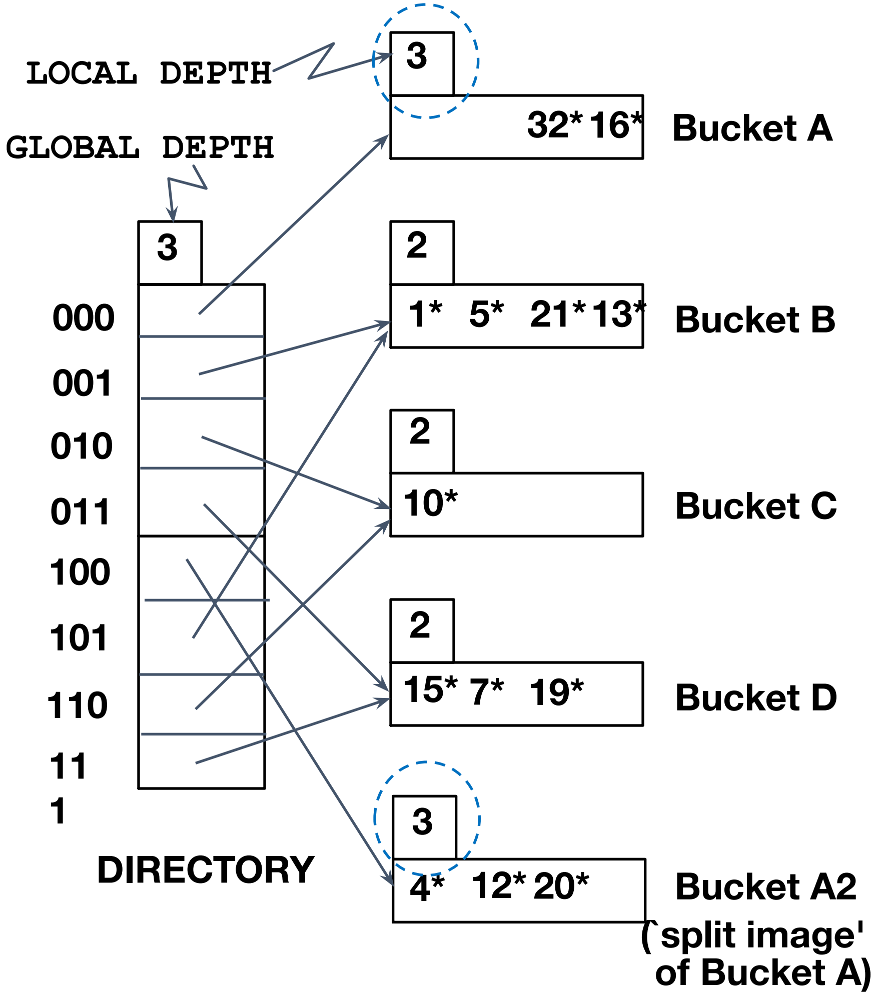


# \Extendible Hashing: Inserting Entries

- Example: insert 20*

Repeat…

FINAL
STATE!



# Some Applications of Hash Tables

**Database systems:** Specifically, those that require efficient random access. Generally, database systems try to optimize between two types of access methods: sequential and random. Hash tables are an important part of efficient random access because they provide a way to locate data in a constant amount of time.

**Symbol tables:** The tables used by compilers to maintain information about symbols from a program. Compilers access information about symbols frequently. Therefore, it is important that symbol tables be implemented very efficiently.

# Some Applications of Hash Tables ( Cont.)

**Data dictionaries:** Data structures that support adding, deleting, and searching for data. Although the operations of a hash table and a data dictionary are similar, other data structures may be used to implement data dictionaries. Using a hash table is particularly efficient.

**Network processing algorithms:** Hash tables are fundamental components of several network processing algorithms and applications, including route lookup, packet classification, and network monitoring.

**Browser Cashes:** Hash tables are used to implement browser cashes.

# ADT Operations

- Data are entered, accessed, modified and deleted through the external interface, which is a “passageway” located partially “in” and partially out of the ADT.
- Only the public functions are accessible through this interface.
- For each ADT operation there is an algorithm that performs its specific task.

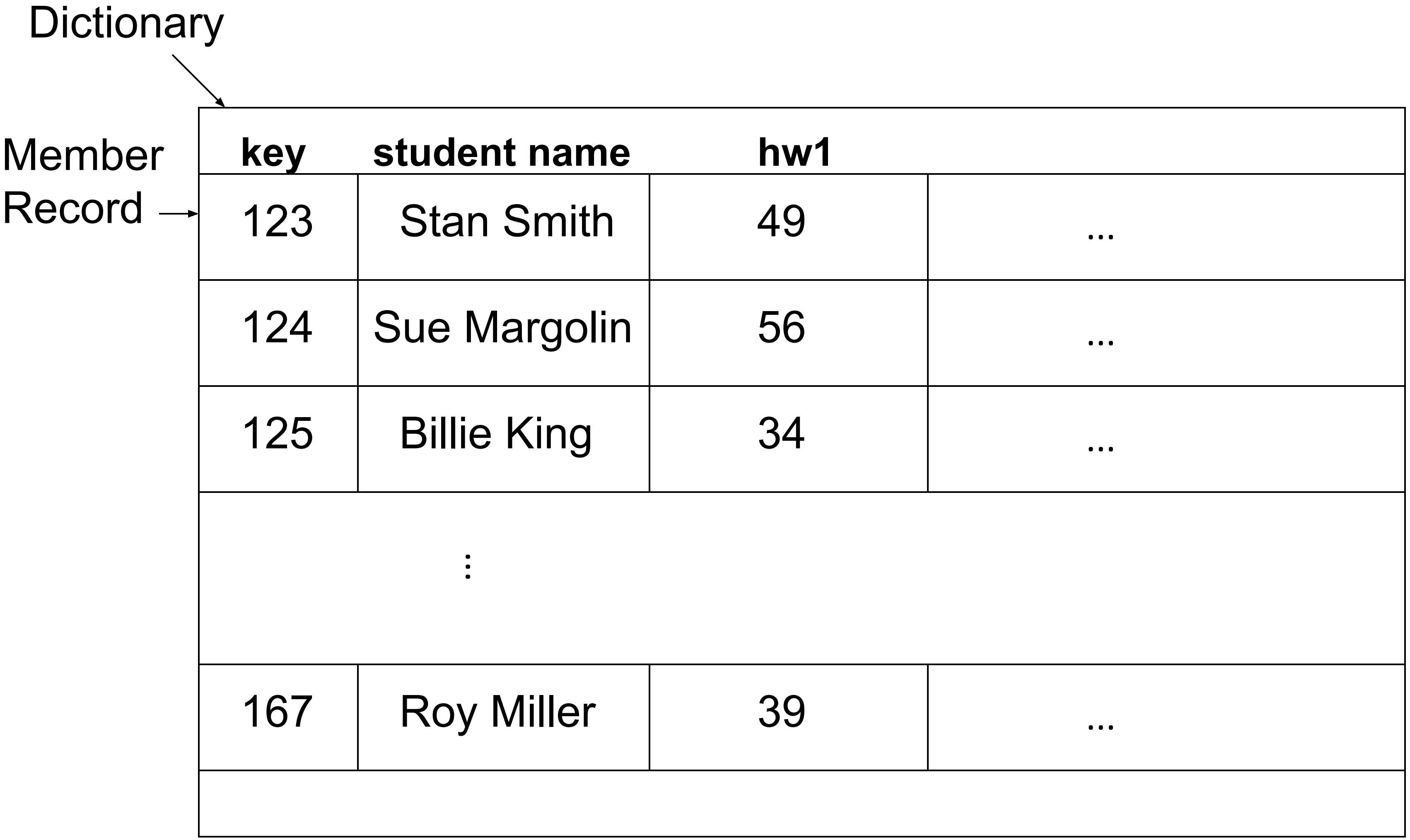
# Typical ADTs:

- Lists
- Stacks
- Queues
- Trees
- Heaps
- Graphs

# Dictionaries

- A dictionary is a collection of elements each of which has a unique search key
    - Uniqueness criteria may be relaxed (multiset)
    - (I.e. do not force uniqueness)
- Keep track of current members, with periodic insertions and deletions into the set
- Examples
    - Membership in a club, course records
    - Symbol table (contains duplicates)
    - Language dictionary (WordSmith, Webster, WordNet)
- Similar to database

# Course Records



Dictionary

Member Record

| key | student name | hw1 | |
|---|---|---|---|
| 123 | Stan Smith | 49 | ... |
| 124 | Sue Margolin | 56 | ... |
| 125 | Billie King | 34 | ... |
| | ⋮ | | |
| 167 | Roy Miller | 39 | ... |
| | | | |

# Dictionary ADT

- simple container methods:**size()**
  **isEmpty()**
  **elements()**

- query methods: **findElement(k)**
  **findAllElements(k)**

- update methods: **insertItem(k, e)**
  **removeElement(k)**
  **removeAllElements(k)**

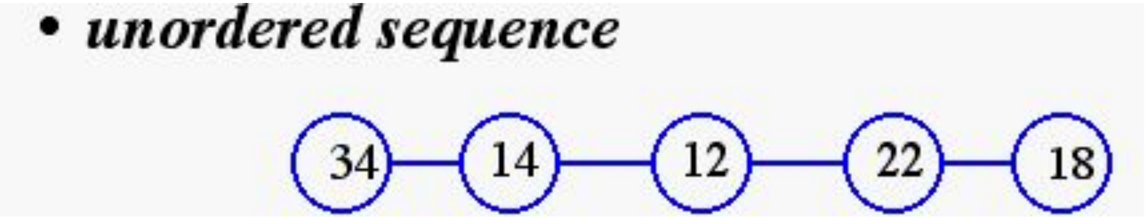
- special element **NO_SUCH_KEY**, returned by an unsuccessful search

# How to Implement a Dictionary?

- Sequences / Arrays
    - ordered
    - unordered
- Binary Search Trees
- Skip lists
- Hashtables

# Recall Arrays …

- Unordered array



- *unordered sequence*

- searching and removing takes O(?) time
- inserting takes O(?) time
- applications to log files (frequent insertions, rare searches and removals)

# More Arrays

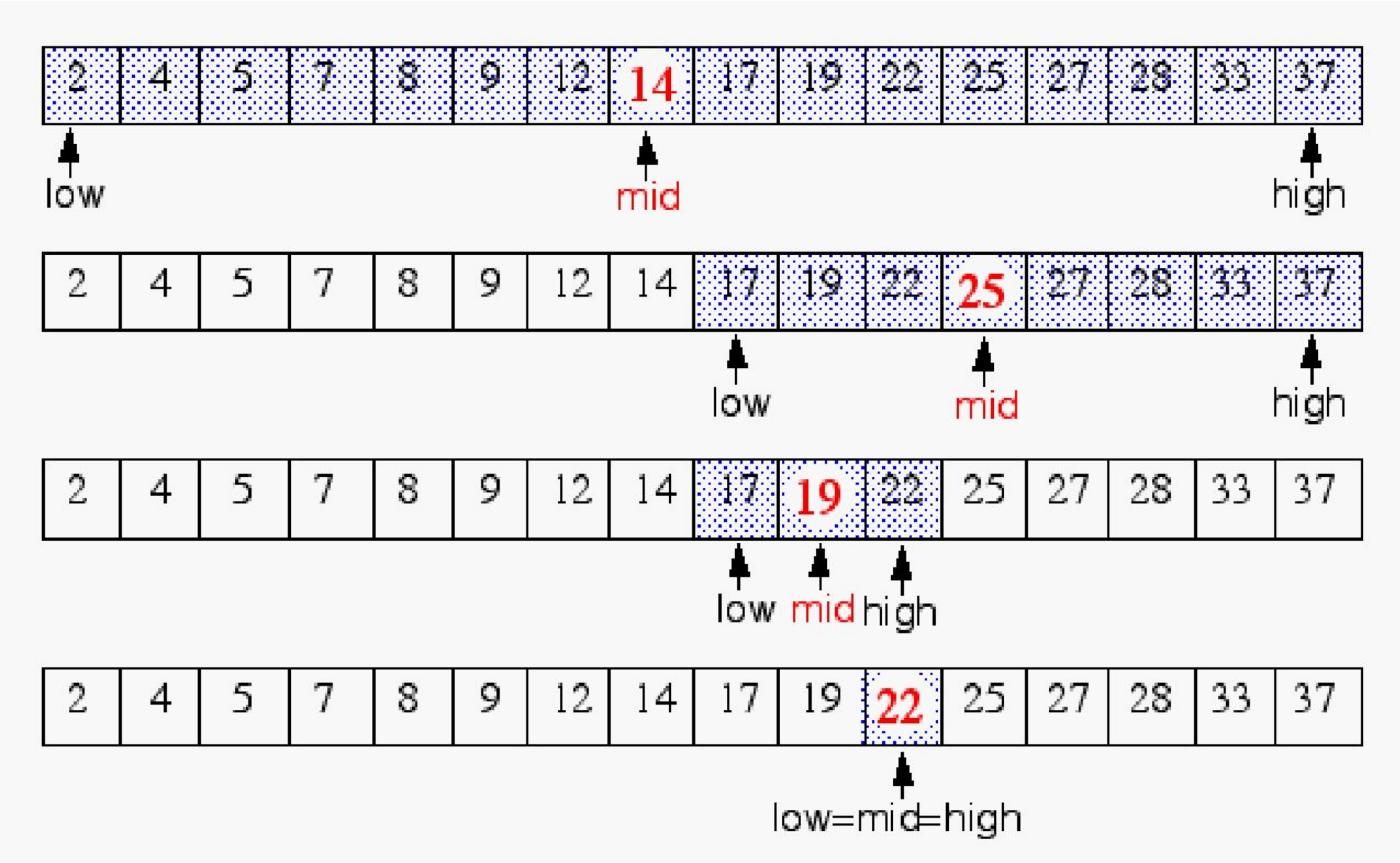
- Ordered array

- searching takes O(log n) time (binary search)
- inserting and removing takes O(n) time
- application to look-up tables (frequent searches, rare insertions and removals)

- Apply binary search

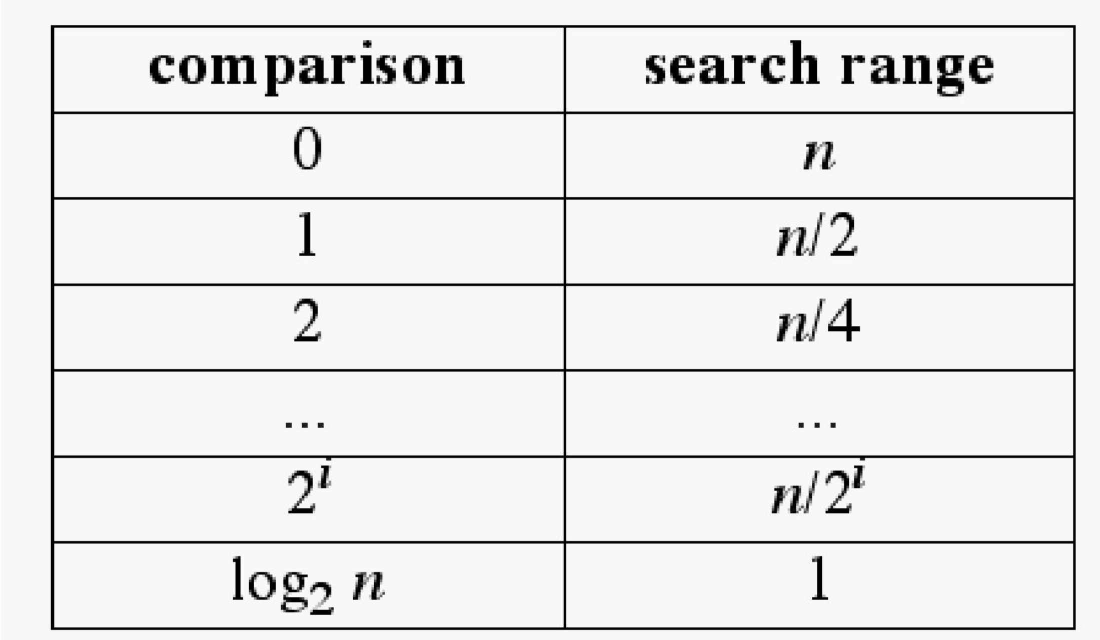
# Binary Searches

- narrow down the search range in stages
- “high-low” game
- findElement(22)



# Running Time of Binary Search

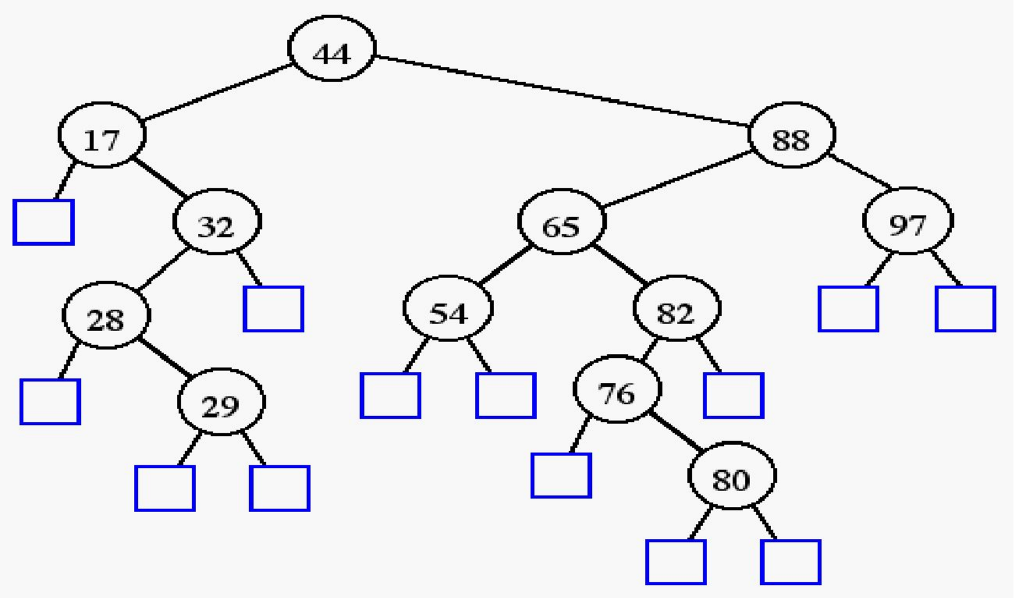
- The range of candidate items to be searched is ***halved after each comparison***



| comparison | search range |
|---|---|
| 0 | $n$ |
| 1 | $n/2$ |
| 2 | $n/4$ |
| ... | ... |
| $2^i$ | $n/2^i$ |
| $\log_2 n$ | 1 |

# Recall Binary Search Trees…

- Implement a dictionary with a BST
  - A binary search tree is a binary tree T such that
  - each internal node stores an item (k, e) of a dictionary.
  - keys stored at nodes in the left subtree of v are less than or equal to k.
  - keys stored at nodes in the right subtree of v are greater than or equal to k.



# SKIP LIST

# Skip Lists

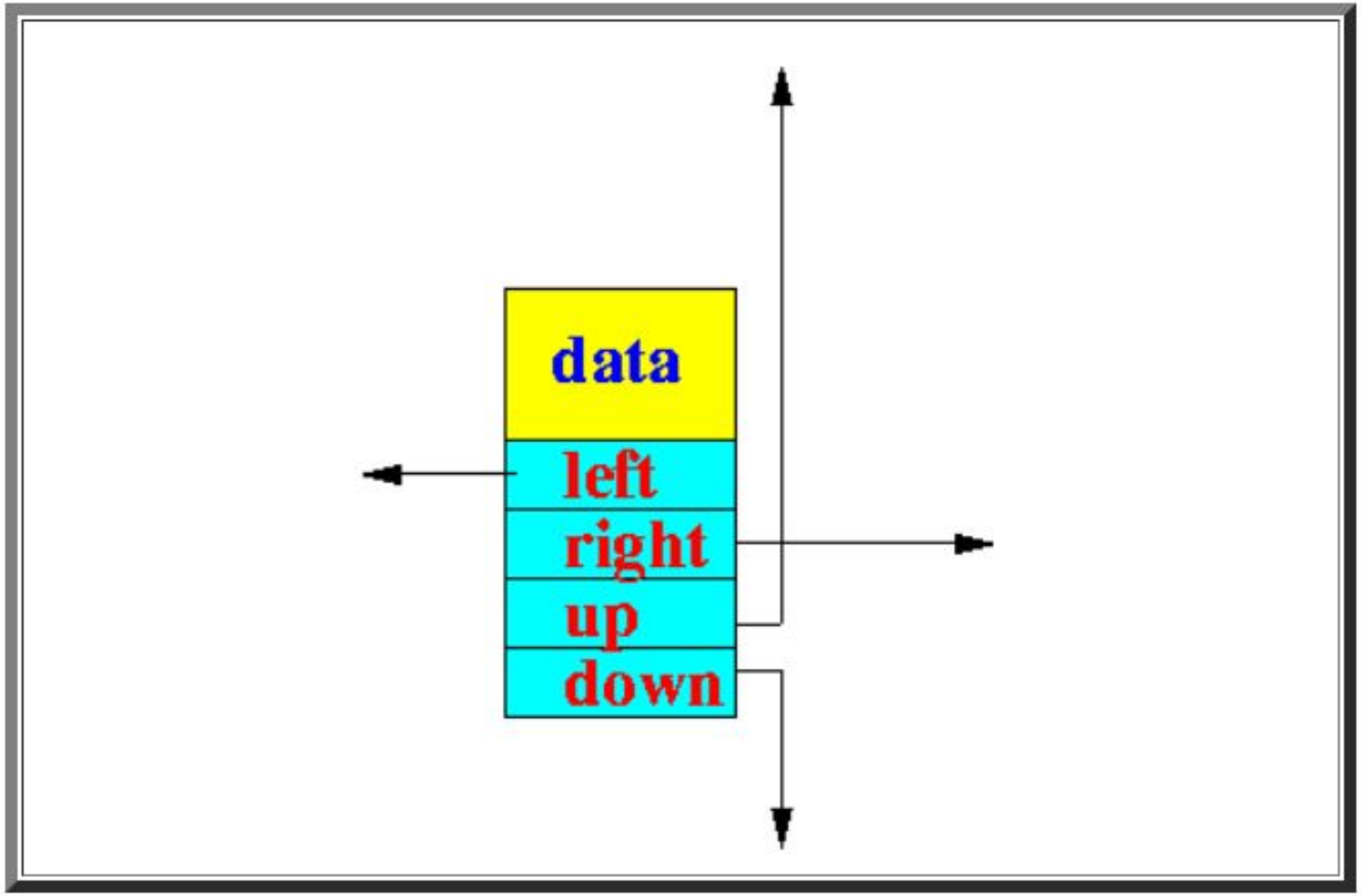
- Definition:
  - A skip list is a probabilistic data structure where elements are kept sorted by key.
  - It allows quick search, insertions and deletions of elements with simple algorithms.
  - It is basically a linked list with additional pointers such that intermediate nodes can be *skipped*.
  - It uses a random number generator to make some decisions.

# Skip Lists

- <u>Skip Levels</u>
  - Doubly Linked lists $S_1..S_h$, each start at $-\infty$ and end at $\infty$ .
  - Level $S_1$ - Doubly linked list containing all the elements in the set S.
  - Level $S_i$ is a subset of level $S_{i-1}$.
  - Each element in Level i has the probability 1/2 to be in level i+1, thus if there are n elements in level $S_1$, the expected number of elements in level $S_i$ is $(n/2)^{i-1}$.
  - The expected number of levels required is O(log n).

## Implementing the skip list data structure

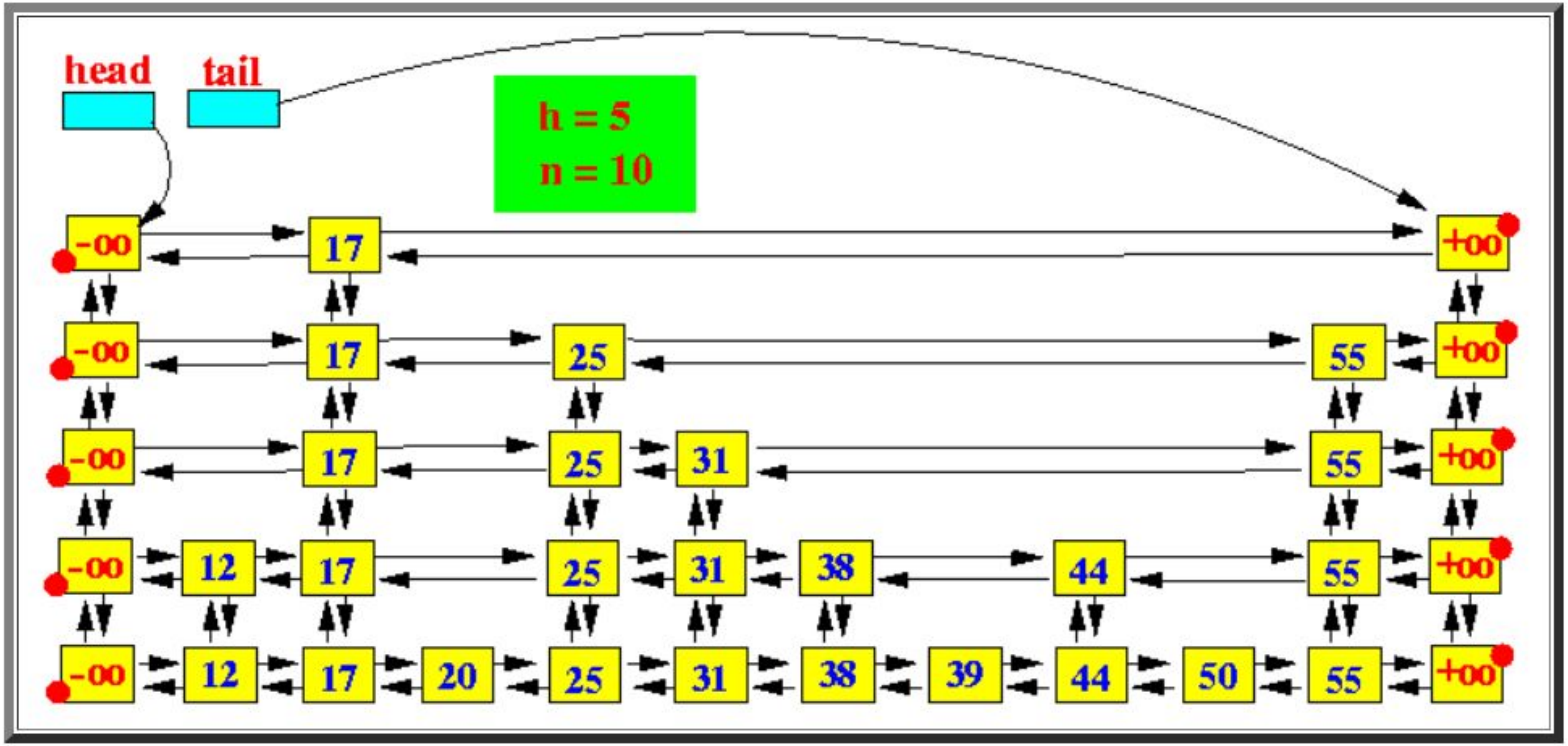
- **The link list element structure used to implement a Skip List**
  - The **link list element** used to implement the **skip list** has **4 links** (not including the **data portion**):



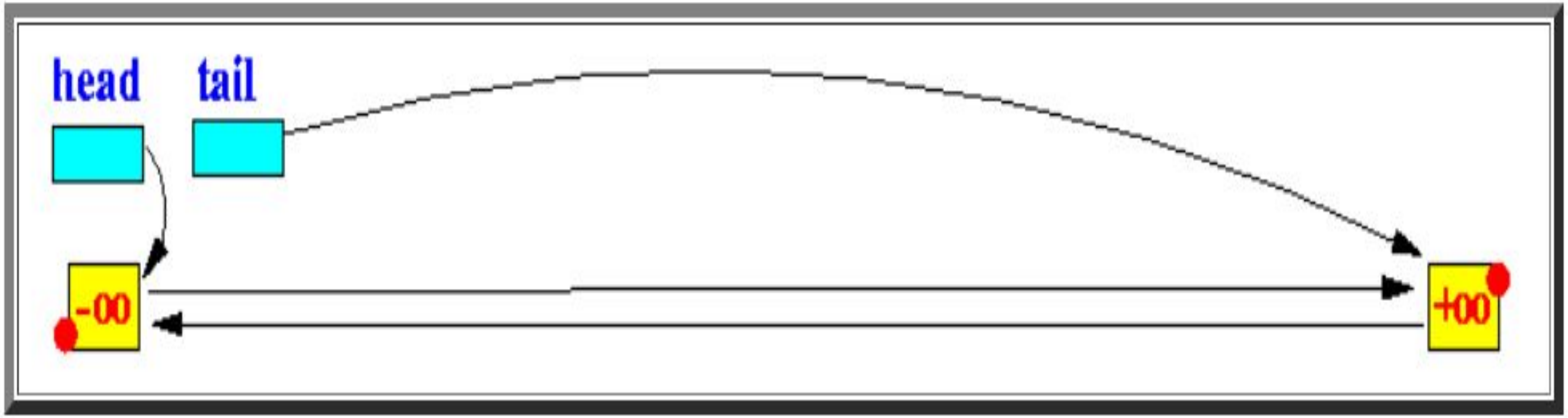
- **The Entry strcuture in a Skip List (the `SkipListEntry` class)**
  - **Skip List entry:**

```
public class SkipListEntry
{
   public String key;
   public Integer value;

   public SkipListEntry up;       // up link
   public SkipListEntry down;     // down link
   public SkipListEntry left;     // left link
   public SkipListEntry right;    // right link

   ...
   (methods)
}
```

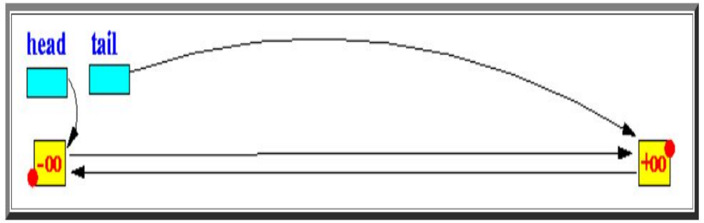
- Example illustrating how the variables are used:



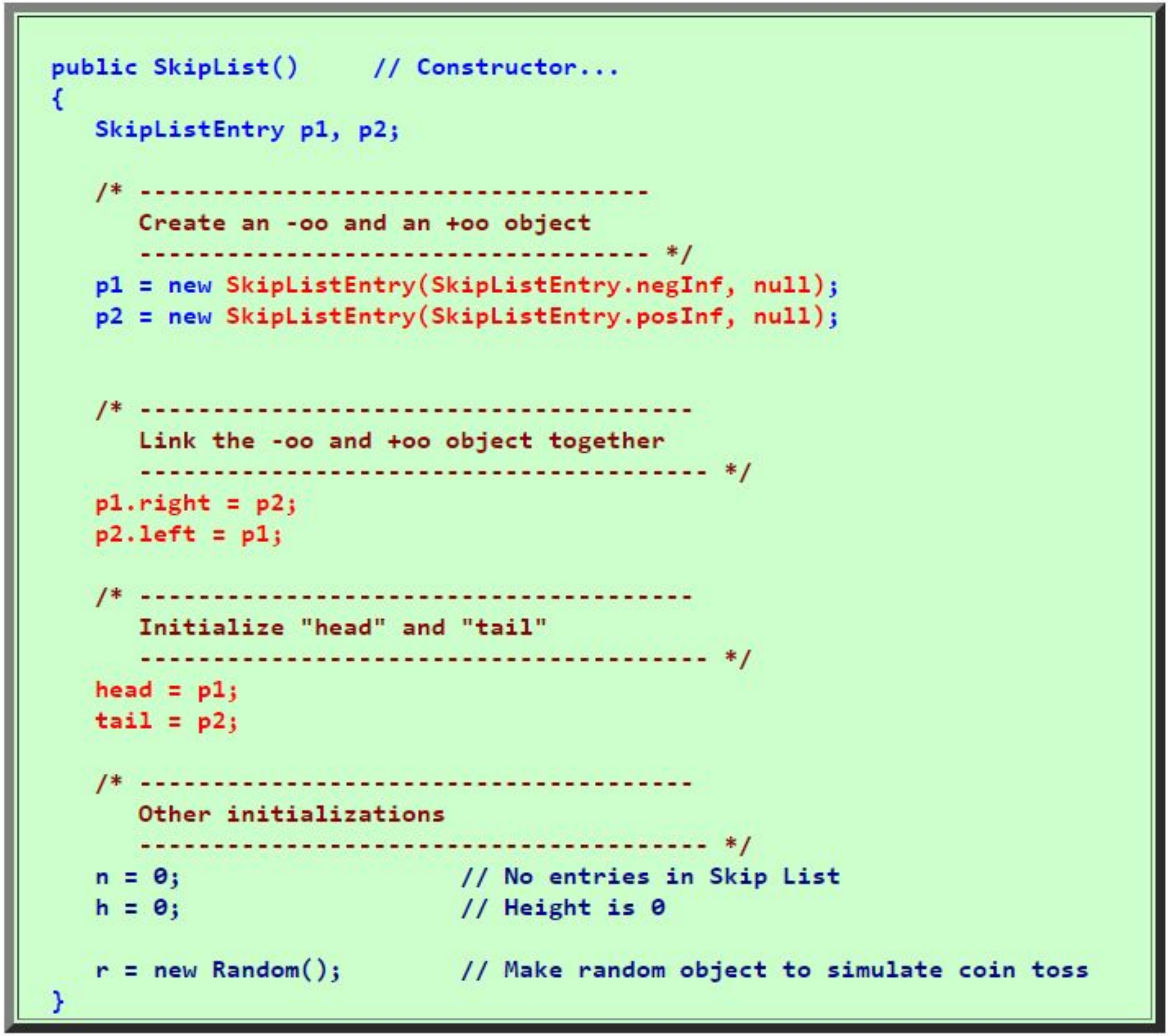
- Constructing a Skip List object
  - The **constructor** will construct an ***empty*** **Skip List** which looks like this:



- Constructing a Skip List object
  - The **constructor** will construct an *empty* Skip List which looks like this:

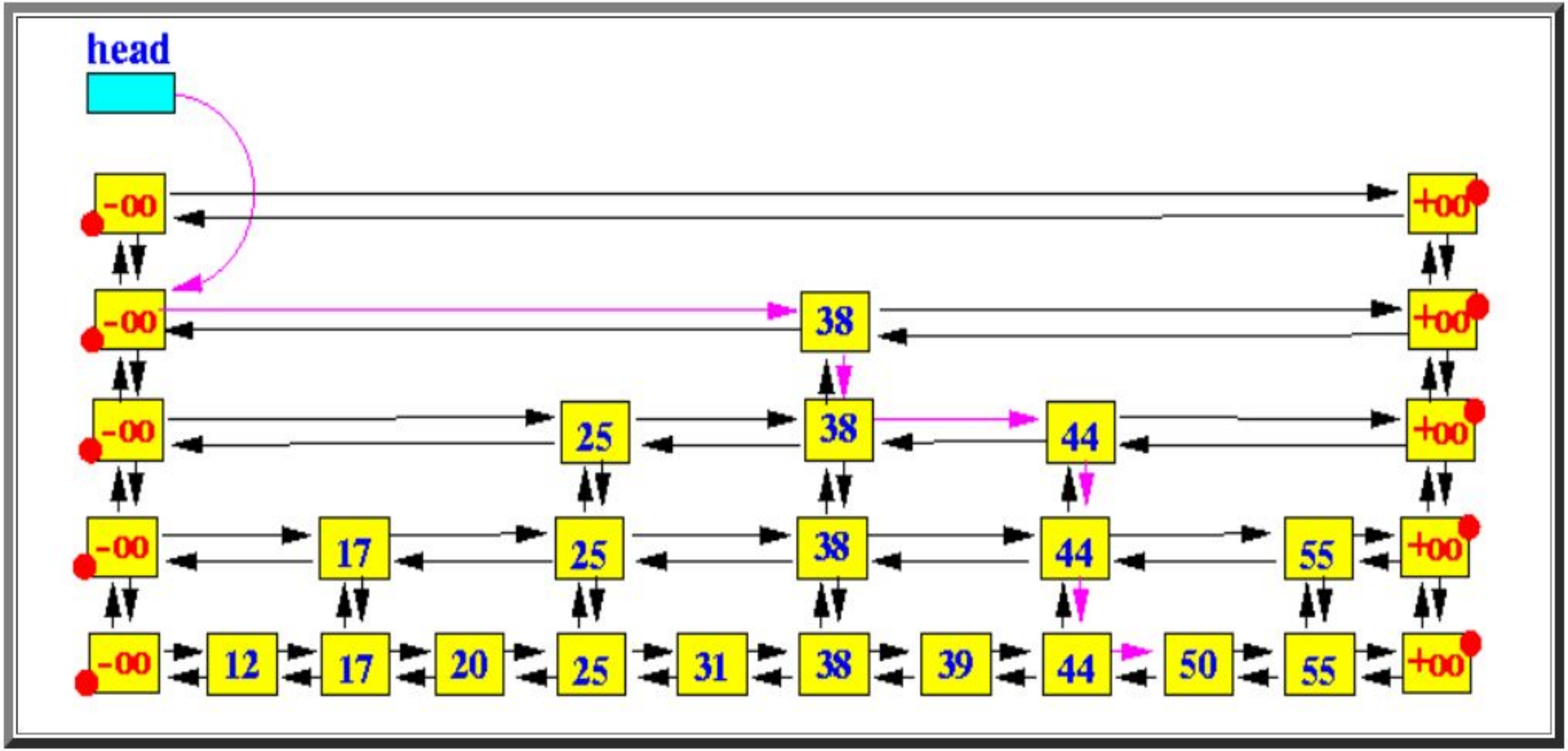


- **Constructor code:**

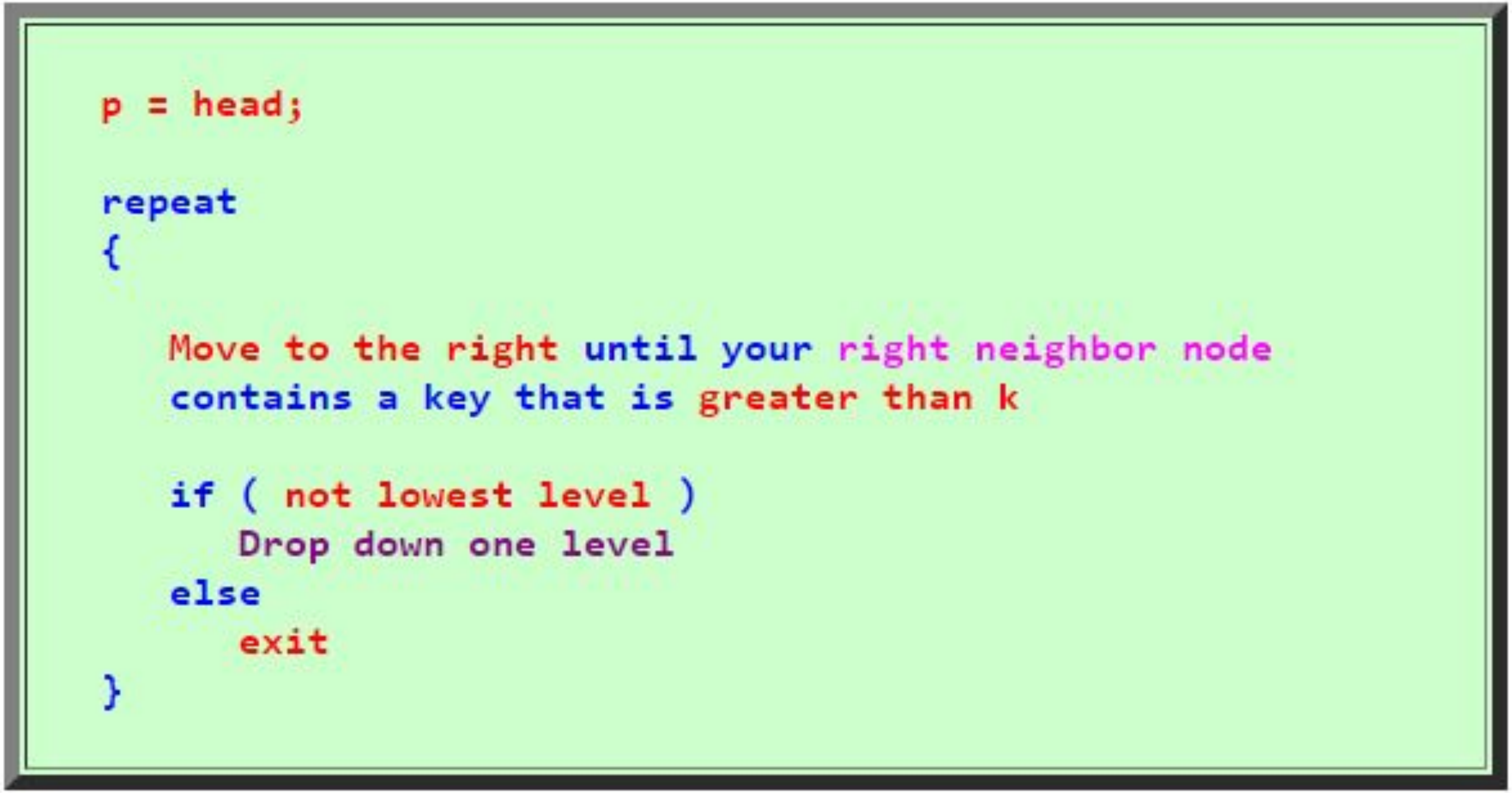


```
public SkipList()     // Constructor...
{
   SkipListEntry p1, p2;

   /* -----------------------------------
      Create an -oo and an +oo object
      ----------------------------------- */
   p1 = new SkipListEntry(SkipListEntry.negInf, null);
   p2 = new SkipListEntry(SkipListEntry.posInf, null);


   /* --------------------------------------
      Link the -oo and +oo object together
      --------------------------------------- */
   p1.right = p2;
   p2.left = p1;

   /* --------------------------------------
      Initialize "head" and "tail"
      --------------------------------------- */
   head = p1;
   tail = p2;

   /* --------------------------------------
      Other initializations
      --------------------------------------- */
   n = 0;                   // No entries in Skip List
   h = 0;                   // Height is 0

   r = new Random();        // Make random object to simulate coin toss
}
```

- **Search operation in a skip list**
  - Consider the **links traversed** to locate the key **50**:



- **Psuedo code:**



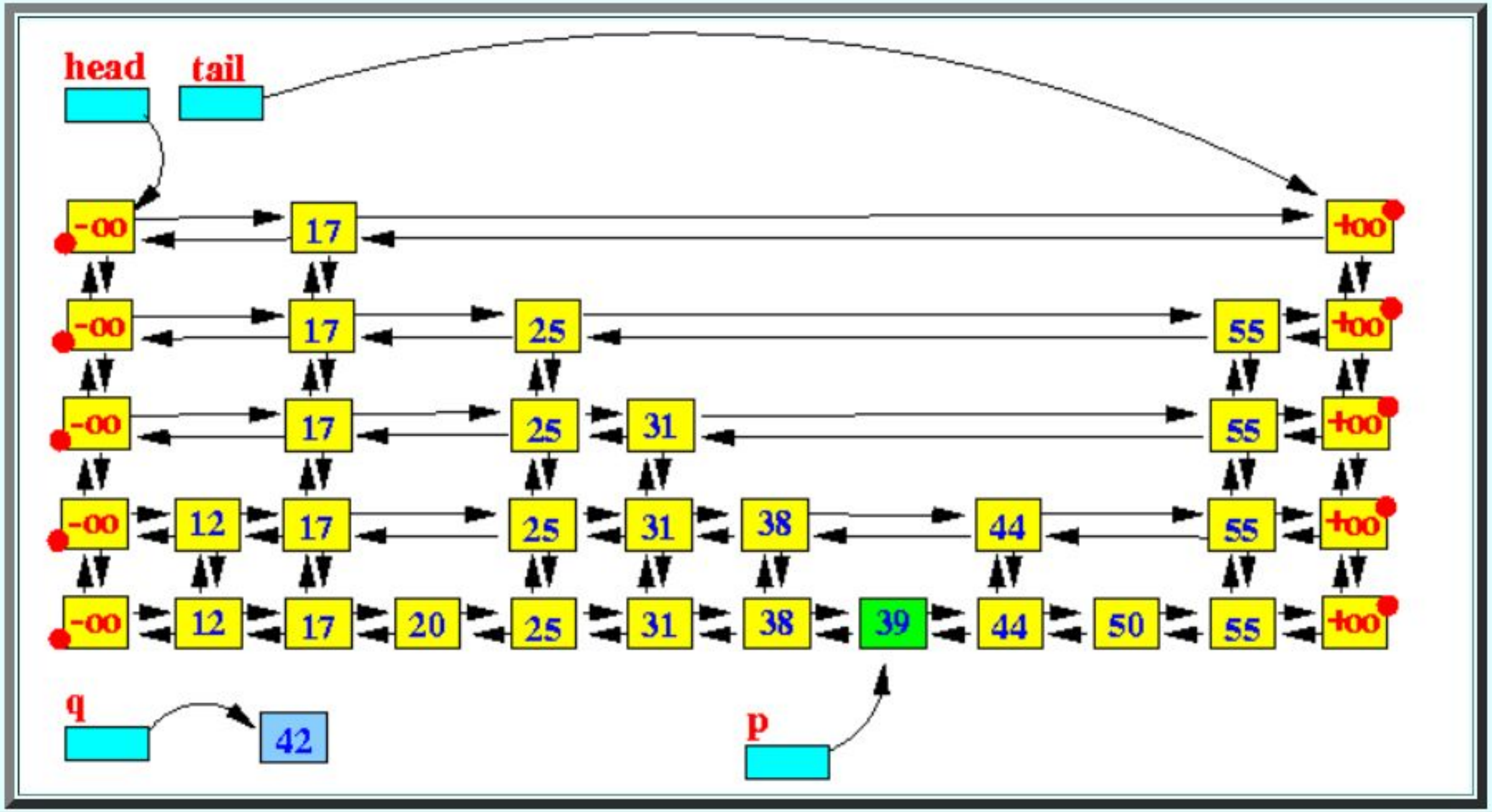
```
p = head;

repeat
{

   Move to the right until your right neighbor node
   contains a key that is greater than k

   if ( not lowest level )
      Drop down one level
   else
      exit
}
```

- If the key **k** is found in the **Skip List**, `findEntry(k)` will return the **reference** to the entry containg the key **k**

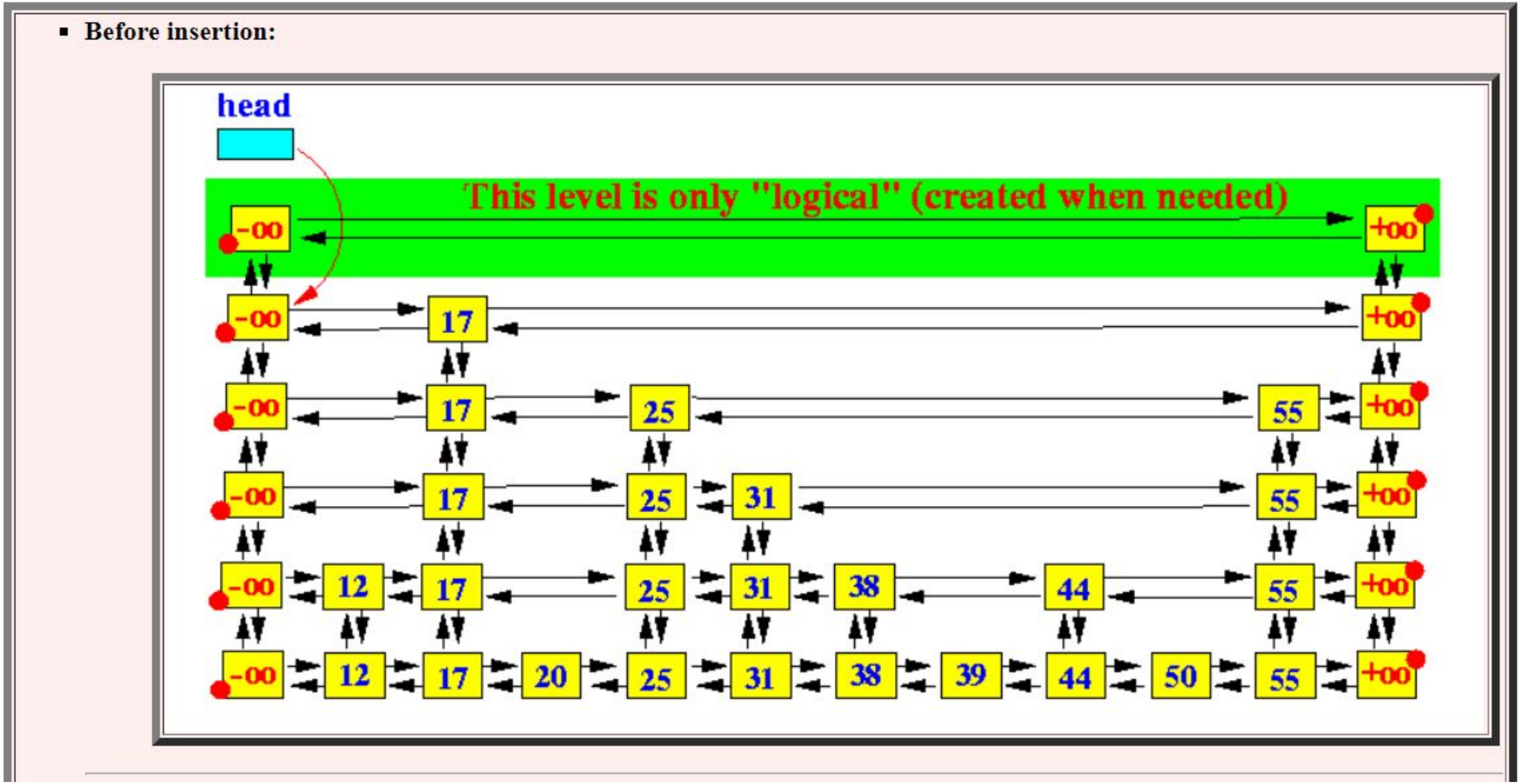
---

- If the key **k** is *not* **found** in the **Skip List**, `findEntry(k)` will return the **reference** to the **floorEntry(k)** entry containg a key that is ***smaller* than k**

**Example:** findEntry(42) will return the reference to **39**:



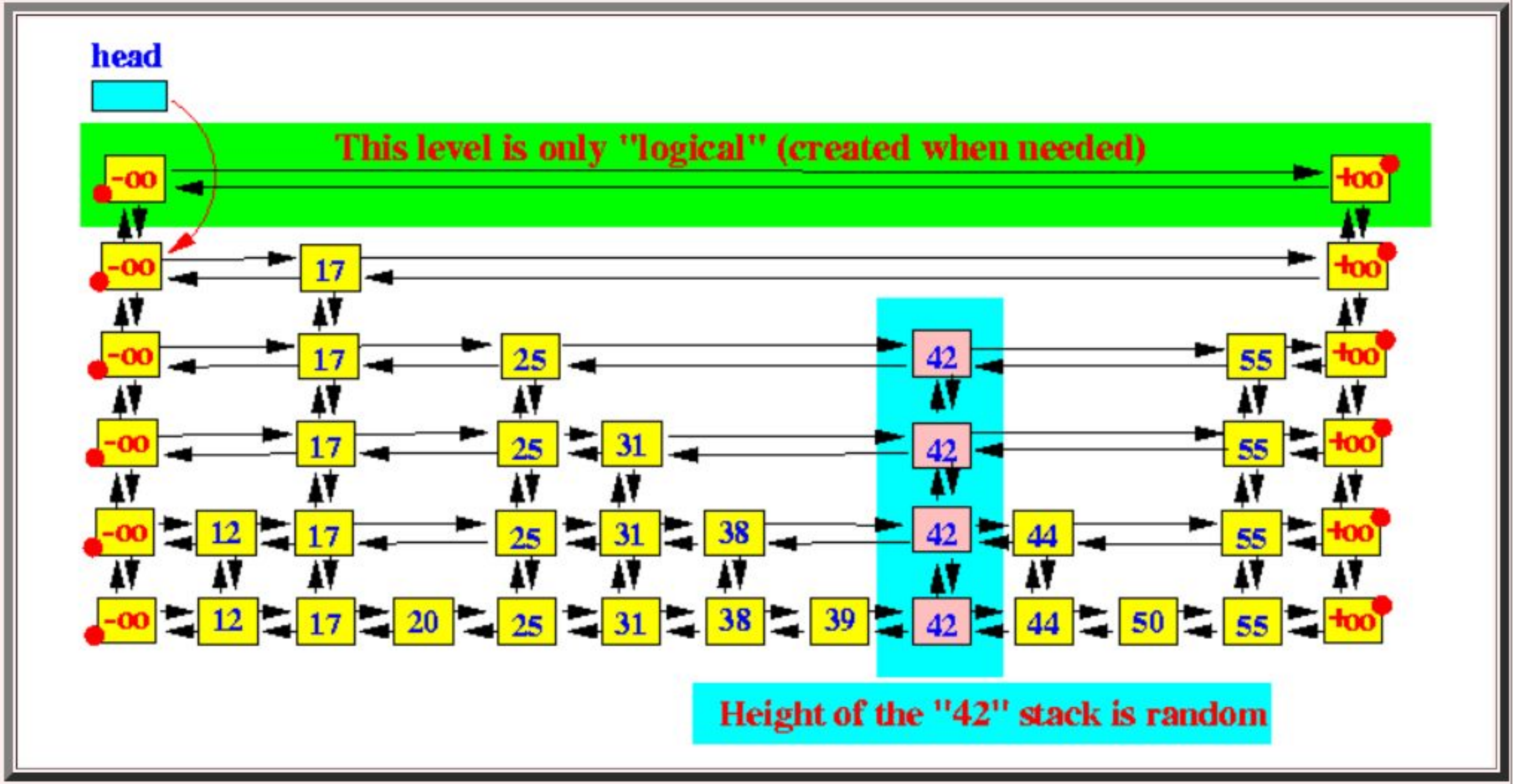
# Insert a New Entry



- **Before insertion:**

head
This level is only "logical" (created when needed)
-oo +oo
-oo 17 +oo
-oo 17 25 55 +oo
-oo 17 25 31 55 +oo
-oo 12 17 25 31 38 44 55 +oo
-oo 12 17 20 25 31 38 39 44 50 55 +oo

- **After inserting** key 42:

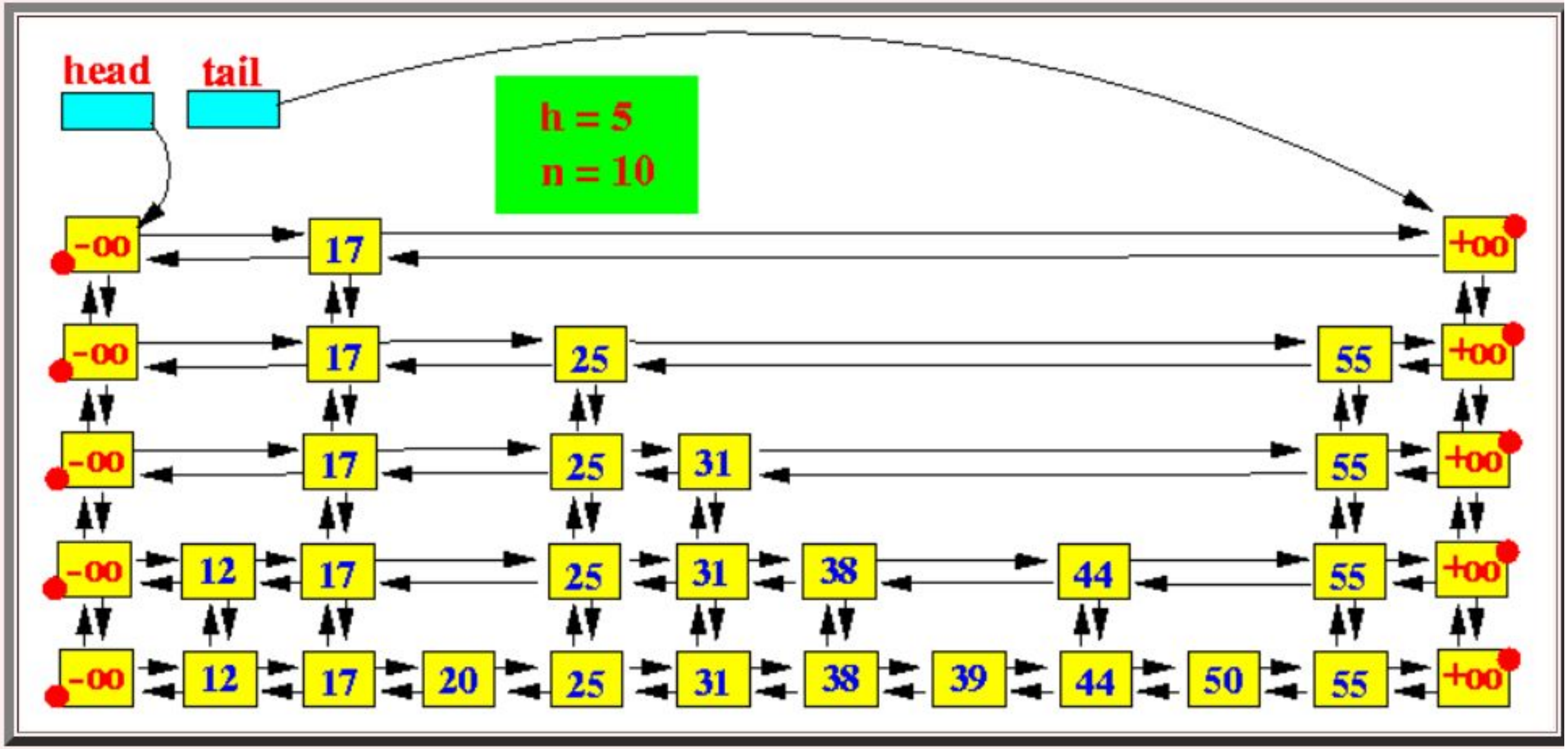

head
-oo +oo
-oo 17 +oo
-oo 17 25 42 55 +oo
-oo 17 25 31 42 55 +oo
-oo 12 17 25 31 38 42 44 55 +oo
-oo 12 17 20 25 31 38 39 42 44 50 55 +oo

This level is only "logical" (created when needed)

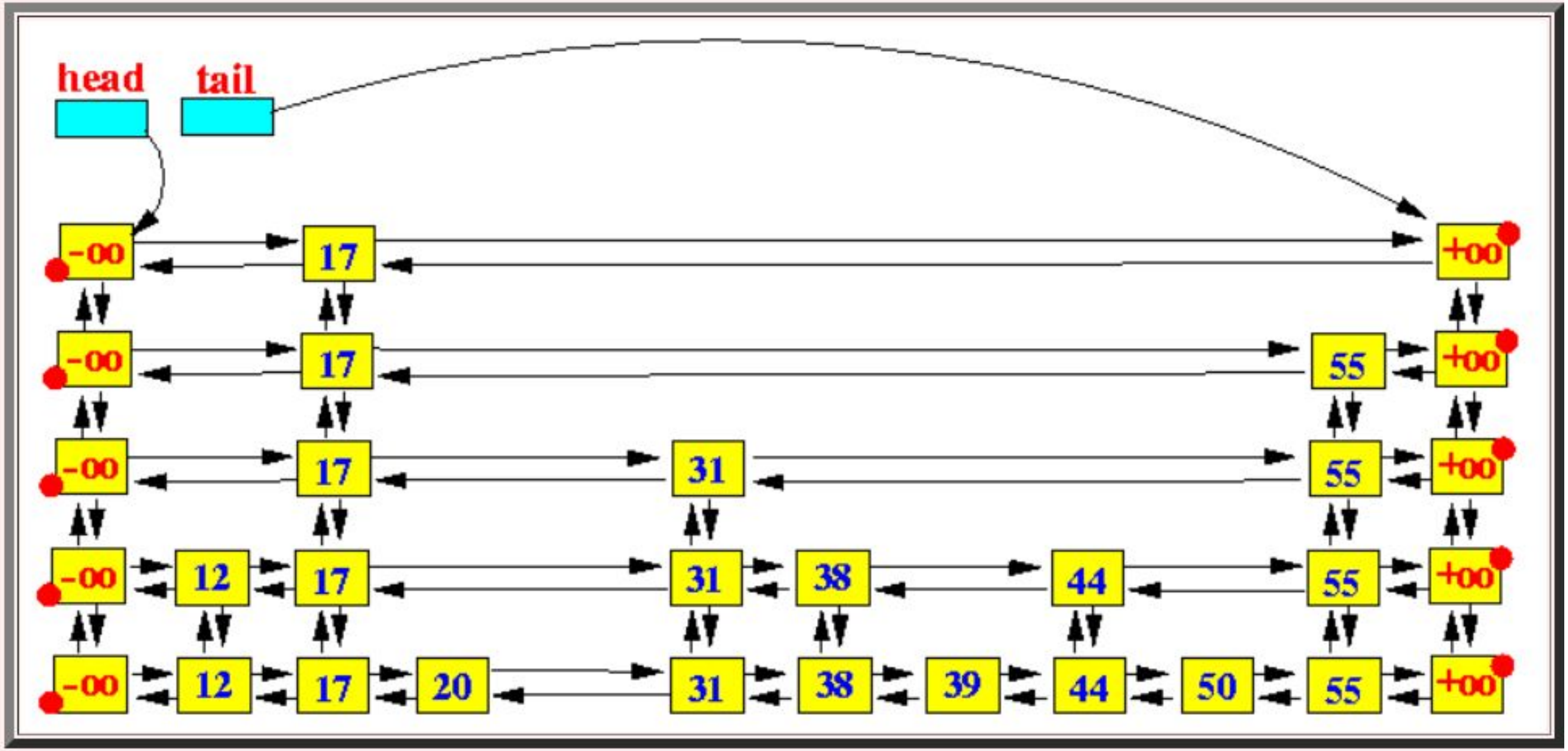
Height of the "42" stack is random

# Deleting an entry from a skip list

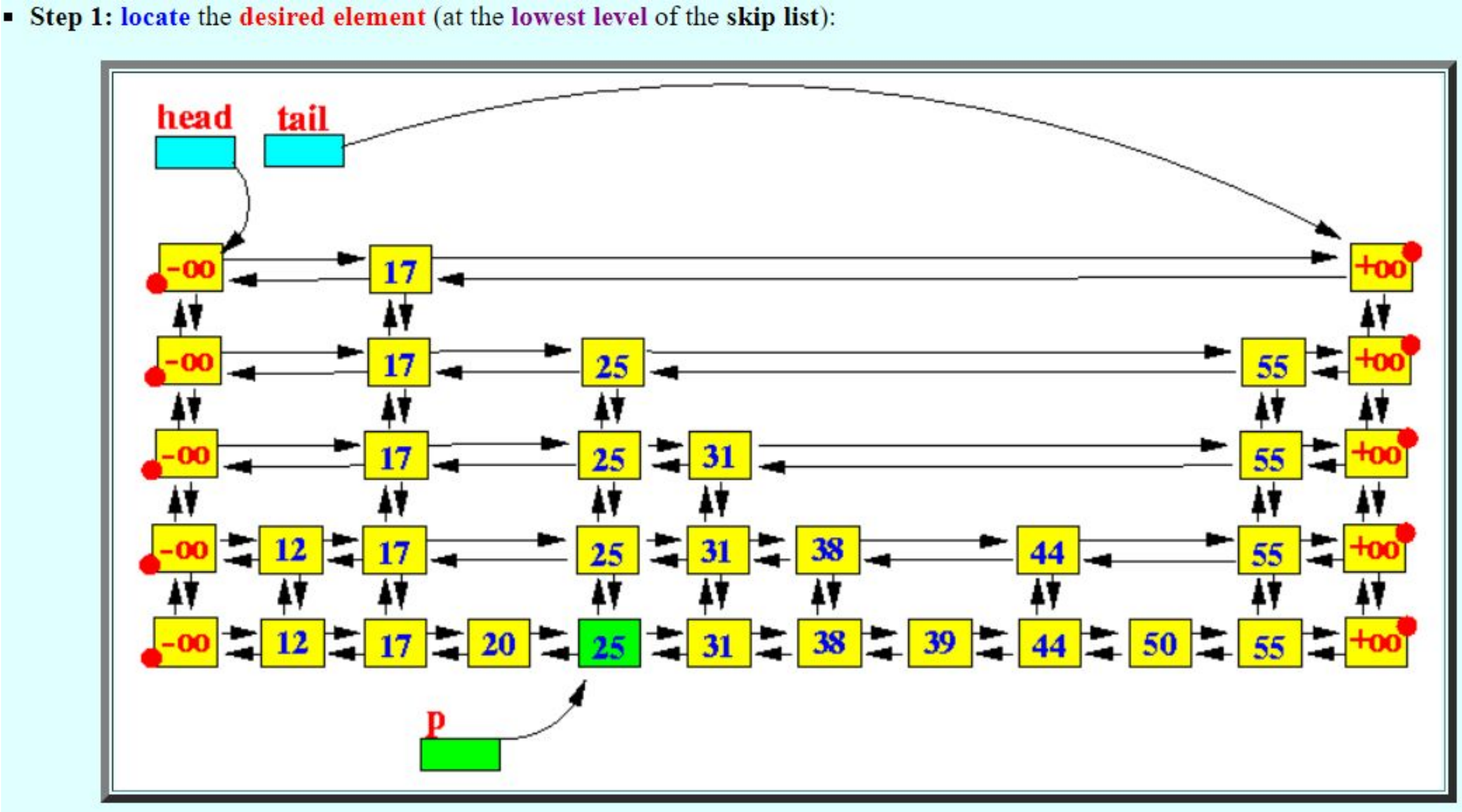
- **Before** deletinng the entry **25**:


head
tail
h = 5
n = 10
-oo 17 +oo
-oo 17 25 55 +oo
-oo 17 25 31 55 +oo
-oo 12 17 25 31 38 44 55 +oo
-oo 12 17 20 25 31 38 39 44 50 55 +oo

- *After* deleting the entry **25**:



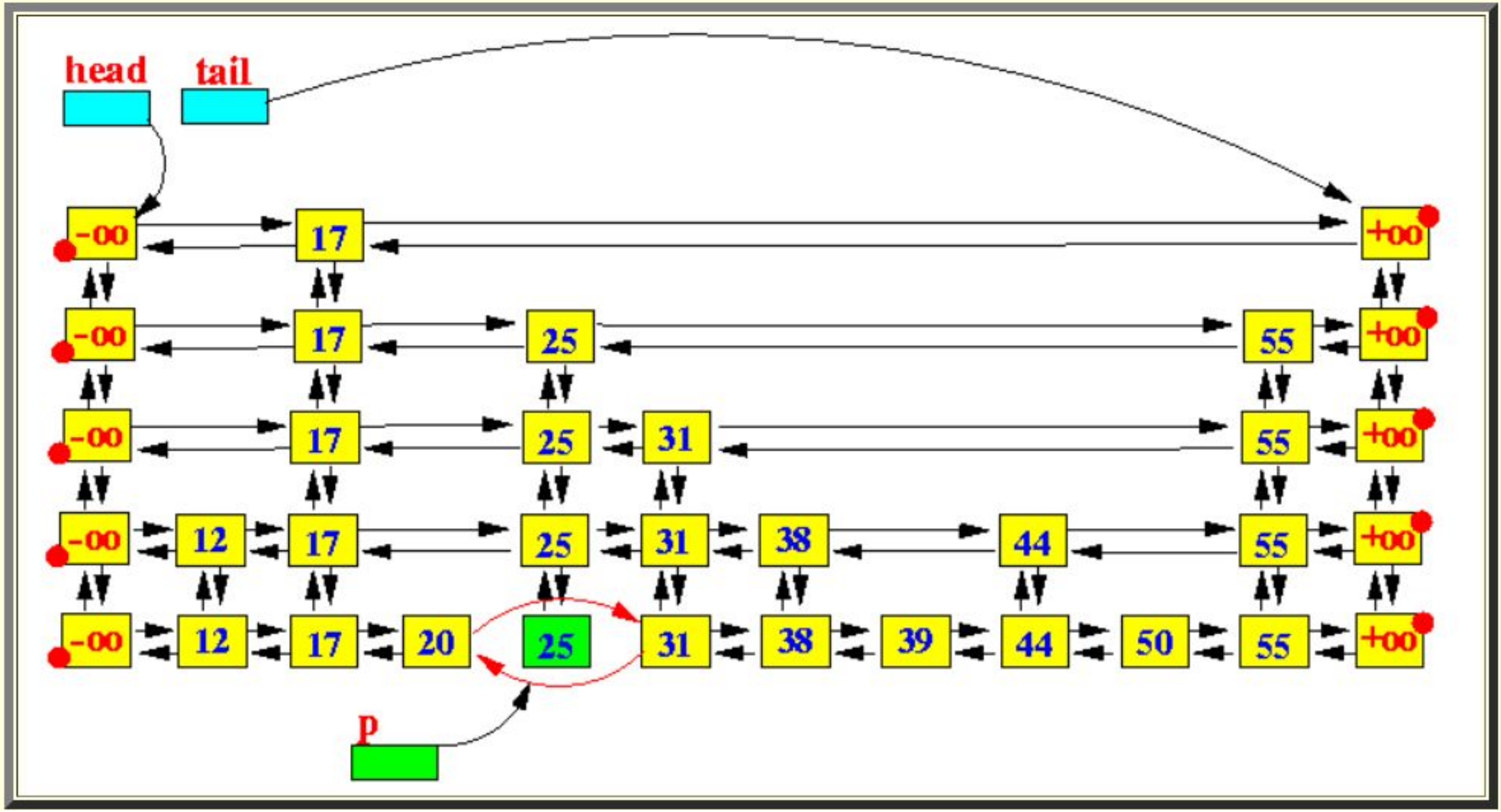
# Step-by-step to accomplish: remove(25)



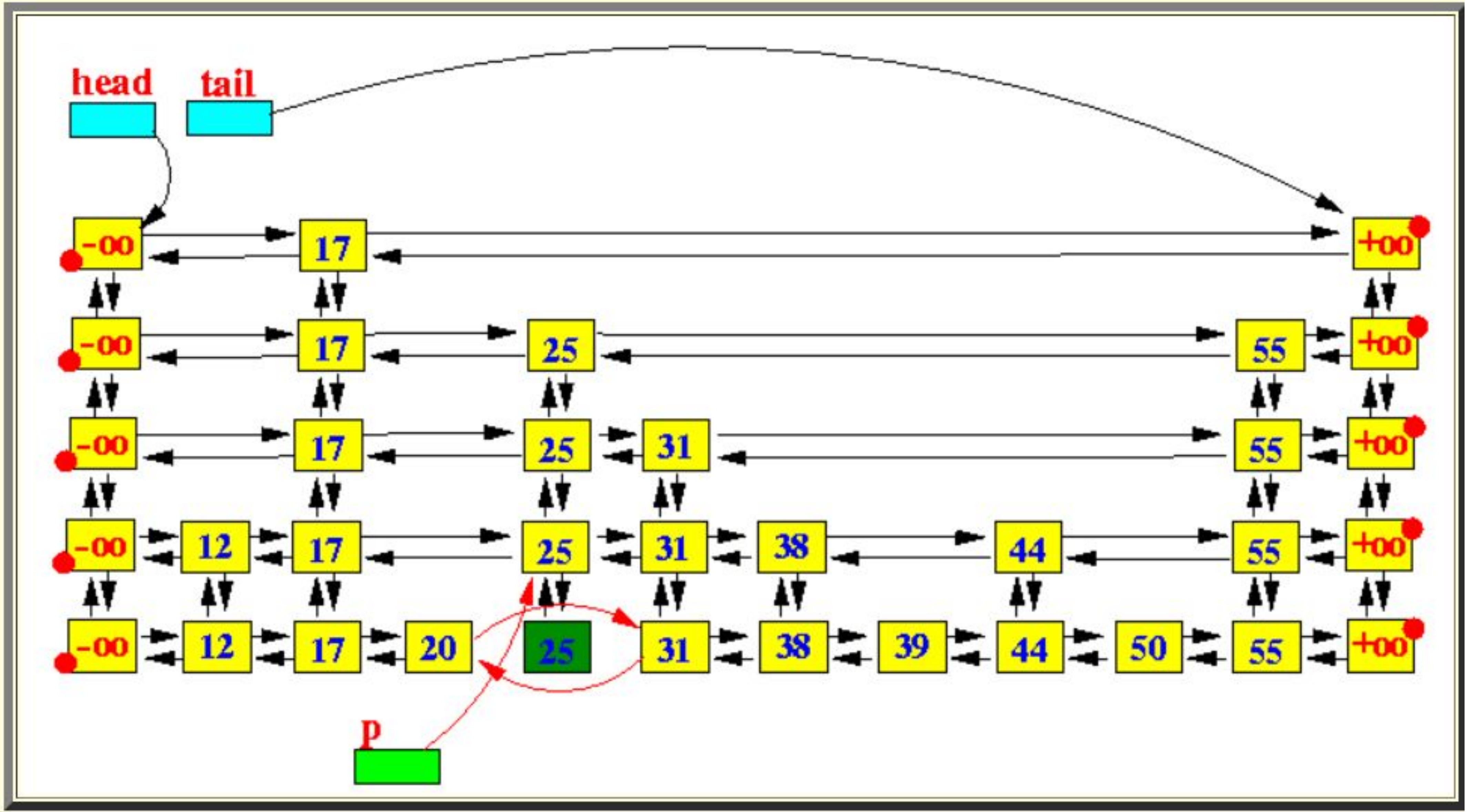
- **Step 1: locate** the **desired element** (at the **lowest level** of the **skip list**):

head
tail
-oo 17 +oo
-oo 17 25 55 +oo
-oo 17 25 31 55 +oo
-oo 12 17 25 31 38 44 55 +oo
-oo 12 17 20 25 31 38 39 44 50 55 +oo
p

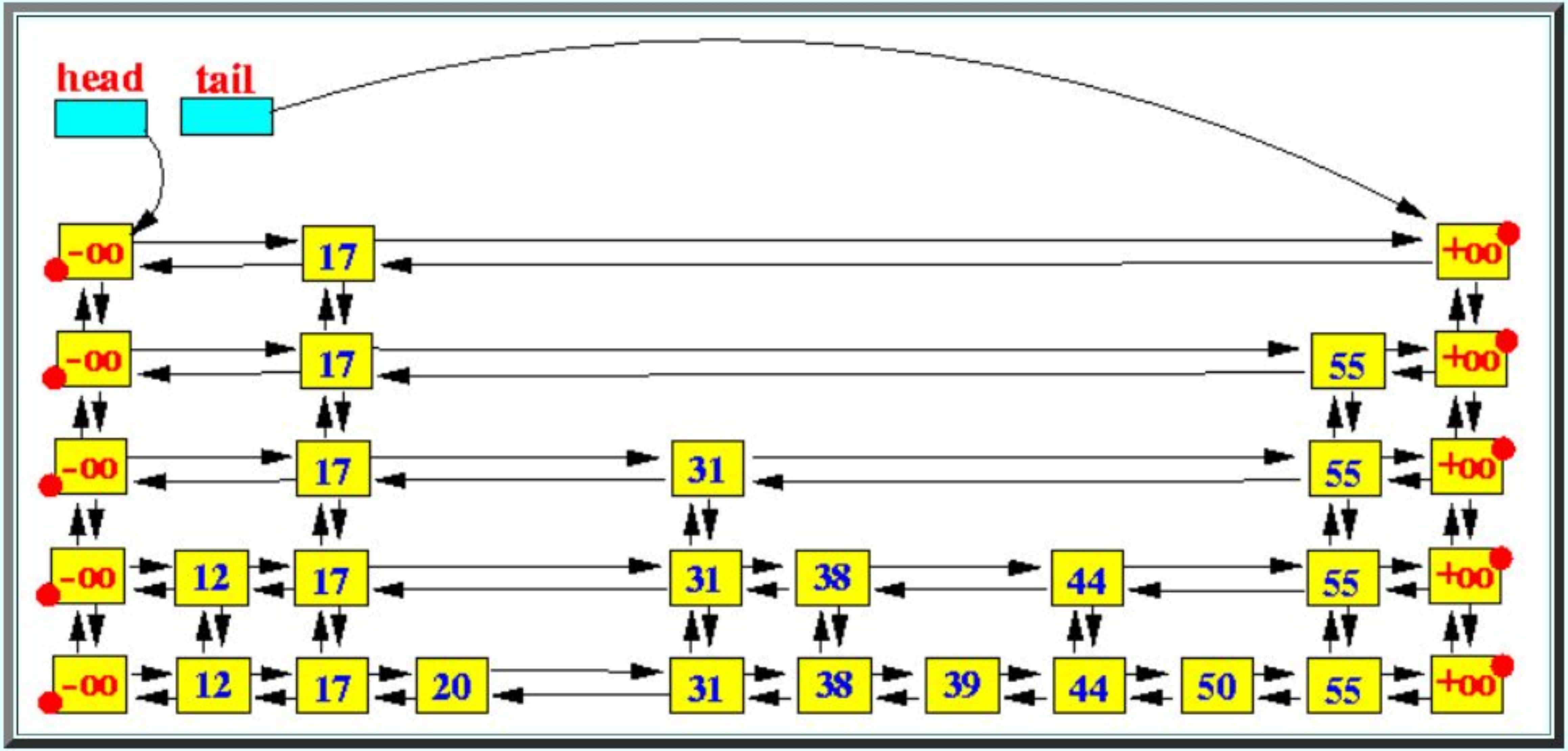
- **While p != null, repeat these steps to remove the column:**
    - **Unlink** the element at **p** (by making the **left neighbor** and the **right neighbor** pointing to **each other**)



- Move **p** ***upward*** (prepare for loop)



- **Result of removal:**


head
tail
-oo
17
+oo
-oo
17
55
+oo
-oo
17
31
55
+oo
-oo
12
17
31
38
44
55
+oo
-oo
12
17
20
31
38
39
44
50
55
+oo