Chapter 9

Hashing

Dr. Hadeel AlObaidy
halobaidy@uob.edu.bh

Outline

1. Introduction
2. Hash Functions
3. Collision Resolution
   a) Open Addressing (Linear Probing, Quadratic Probing)
   b) Chaining
4. Exercise

1. Introduction

- 2 algorithms are used to search an item x in an array of size n:
  - **Sequential**: compare x item by item (complexity n).
  - **Binary**: divide the array into two equidistant parts, compare x with middle item, and continue the search in one of the two halves of the array depending on whether x is equal, greater or less than the middle item (complexity $\log_2 n$).
- The two above algorithms are comparison-based algorithms.
- For the binary search algorithm, the items must be sorted.
- Can we construct an algorithm with complexity less than $\log_2 n$? .

1. Introduction

- The search algorithm that we now describe is called **hashing**.
- Hashing requires the data to be organized via a table called hash table **HT** (stored in an array).
- To search for an item *x* in the table, we apply a function *h*, called **hash function** (compute *h*(*x*)).
- The function h is an arithmetic function and *h*(*x*) gives the address of the item in the hash table.
- If the size of the hash table is *m*, then $0 \le h(x) \le m-1$.
- The items of HT are stored in no particular order.

1. Introduction

- The hash table is usually divided into b **buckets** HT[0], HT[1], …, HT[b-1].
- Each bucket can hold r items ($b \times r = m$ where m is the HT size).
- The function h maps the item x onto an int t: $h(x) = t$ $(0 \leq t \leq b-1)$.
- Two items x_1 and x_2 ($x_1 \neq x_2$) are called **synonyms** if $h(x_1) = h(x_2)$.
- Let x be an item (key) and $h(x) = t$. If bucket t is full, we say that an **overflow** occurs.
- Let x_1 and x_2 ($x_1 \neq x_2$) be two items. If $h(x_1) = h(x_2)$, we say that a **collision** occurs. (if r = 1, an overflow and a collision occur).

2. Hash Functions: Some Examples

- **Mid-Square**: the *h* function is computed by squaring the identifier, and then using the appropriate number of bits from the middle of the square to obtain the bucket address.
- **Folding**: The key *x* is partitioned into parts such that all the parts (except possibly the last part) are of equal length. The parts are then added in some way to obtain the hash address.
- **Division (Modular arithmetic)**: in this method, x is converted into an integer i_x. i_x is then divided by the size of HT to get the reminder (giving the address of *x* in HT)

2. Hash Functions: Division (Modular arithmetic)

- $h(x) = i_x \% HT_{Size}$
- If the key *x* is a string, the following C++ function uses the division method to compute the address of the key:

```
int hasFunction(char *key, int keyLength)
{
    int sum = 0;
    for(int j = 0; j<= keyLength; j++)
        sum += static_cast<int>(key[j]);
    return (sum % HTSize);
}
```

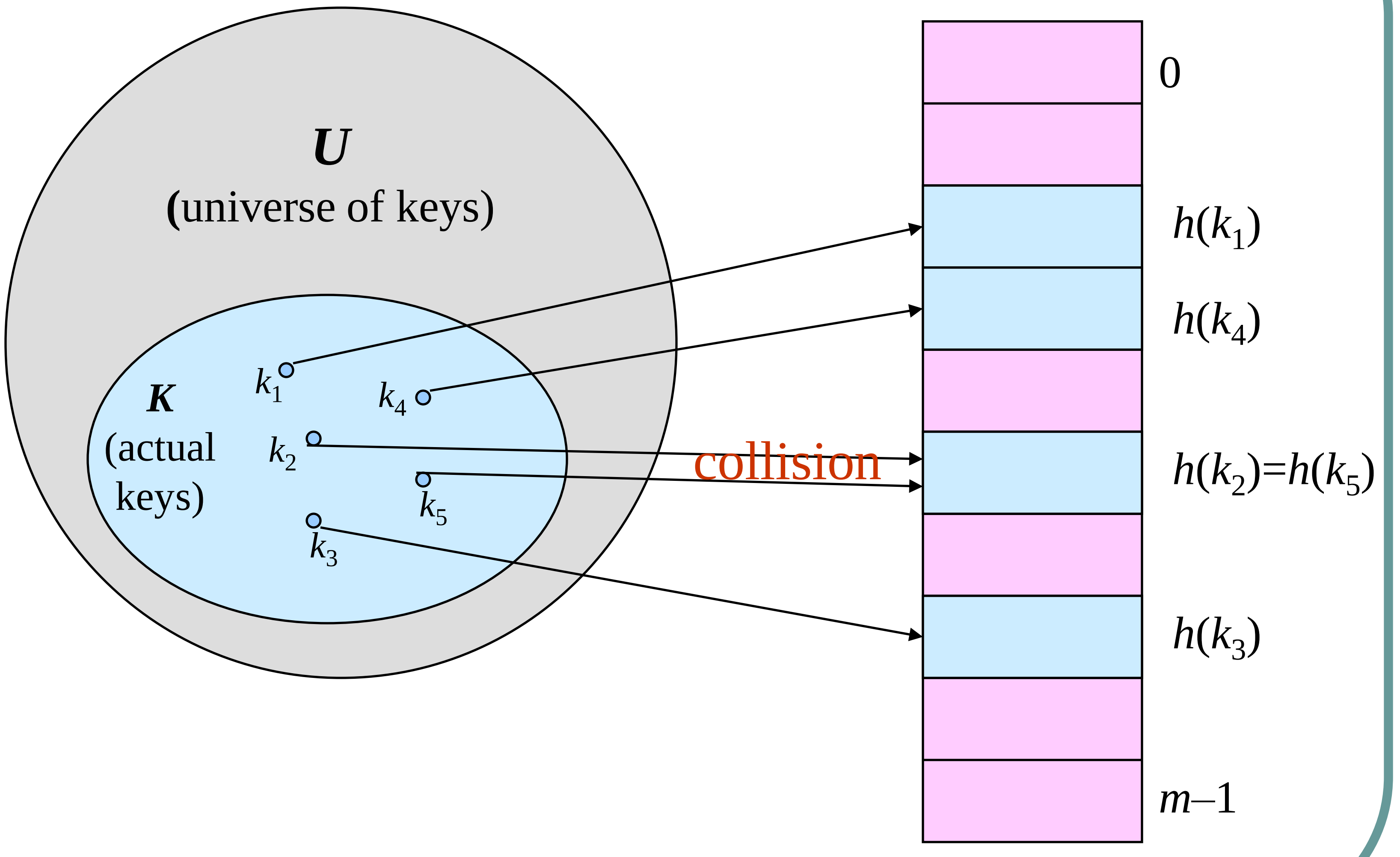
Hashing

- Hash function h: Mapping from U to the slots of a hash table $T[0..m-1]$.

  $$h : U \rightarrow \{0,1,\ldots, m-1\}$$

- With arrays, key k maps to slot $A[k]$.
- With hash tables, key k maps or "hashes" to slot $T[h[k]]$.
- $h[k]$ is the *hash value* of key k.

Hashing



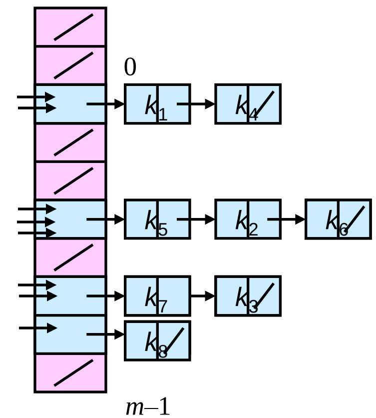
U
(universe of keys)
K
(actual keys)
k1
k4
k2
k5
k3
collision
0
h(k1)
h(k4)
h(k2)=h(k5)
h(k3)
m–1

3. Collision Resolution

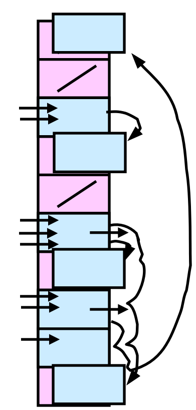
- When choosing a hash function, the main objectives are:
  - Choose a hash function which is easy to compute.
  - Minimize the number of collisions.
- In hashing, we must include algorithms to handle collisions.
- Collision resolution techniques are classified into two categories:
  - **Open addressing (called also closed hashing): Arrays.**
  - **Chaining (called also open hashing): linked lists.**

3. Collision Resolution

There are two kinds of collision resolution:



- Chaining:
  - Store all elements that hash to the same slot in a linked list.
  - Store a pointer to the head of the linked list in the hash table slot.



- Open Addressing:
  - All elements stored in hash table itself.
  - When collisions occur, use a systematic (consistent) procedure to store elements in free slots of the table.

3. Collision Resolution: Open addressing – Linear Probing

- Suppose that an item *x* is to be inserted in HT.
- We use the hash function to compute $h(x)$ the index where the item is likely to be stored.
- Suppose $h(x) = t$, then $0 \leq t \leq HT_{Size}-1$).
- If HT[t] is already occupied, there fore we have a collision.
- In linear probing, starting at location t, we search the array sequentially to find the next available array slot.
- We assume that the array is circular (use of % operation).

3. Collision Resolution: Open addressing – Linear Probing

- Starting at location t, we check the array locations t, (t+1)%HTSize, (t+2)%HTSize, …, (t+j)%HTSize.
- This is called probe sequence.
- The next array slot is given by: (h(x)+j)%HTSize where j is the j^{th} probe.

3. Collision Resolution: Open addressing

- Starting at location t, we check the array locations t, (t+1)%HTSize, $(t+2^2)$%HTSize, $(t+3^2)$%HTSize, …,$(t+j^2)$%HTSize.
- **Example #1:**
- HTSize = 101, $h(X_1) = 25$, $h(X_2) = 96$, and $h(X_3) = 34$.
- The probe sequence for X_1 is: 25, 26, 29, 34, 41, …
- The probe sequence for X_2 is: 96, 97, 100, 4, 11, …
- The probe sequence for X_3 is: 34, 35, 38, 43, 50, 59, …

Example #2

- HTSize =8, keys *a,b,c,d* have hash values h(a)=3, h(b)=0, h(c)=4, h(d)=3
- **Where do we insert *d*?** 3 already filled
- Probe sequence using linear hashing:
  - $h_1(d) = (h(d)+1)\%8 = 4\%8 = 4$
  - $h_2(d) = (h(d)+2)\%8 = 5\%8 = \mathbf{5}^*$
  - $h_3(d) = (h(d)+3)\%8 = 6\%8 = 6$
  - etc.
  - 7, 0, 1, 2

0	b
1	
2	
3	a
4	c
5	**d**
6	
7	

Example #3

- Example:
  - ***h***(***x***) = ***x*** mod 13
  - h(x,i)=(h(x) + i) mod 13
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

0	1	2	3	4	5	6	7	8	9	10	11	12

		41			18	44	59	32	22	31	73	
0	1	2	3	4	5	6	7	8	9	10	11	12

Quadratic Probing

- $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$ $\quad c_1 \neq c_2$

key Probe number Auxiliary hash function

- The initial probe position is $T[h'(k)]$, later probe positions are offset by amounts that depend on a quadratic function of the probe number *i*.
- Must constrain c_1, c_2, and m to ensure that we get a full permutation of $\langle 0, 1, \ldots, m-1 \rangle$.
- Can suffer from ***secondary clustering***:
  - If two keys have the same initial probe position, then their probe sequences are the same.

3. Collision Resolution: Open addressing – Quadratic Probing

- Quadratic probing does not probe all the positions in the table.

$2^2 = 1 + (2\times2 - 1)$

$3^2 = 1 + 3 + (2\times3 - 1)$

$5^2 = 1 + 3 + 5 + (2\times4 - 1)$

…

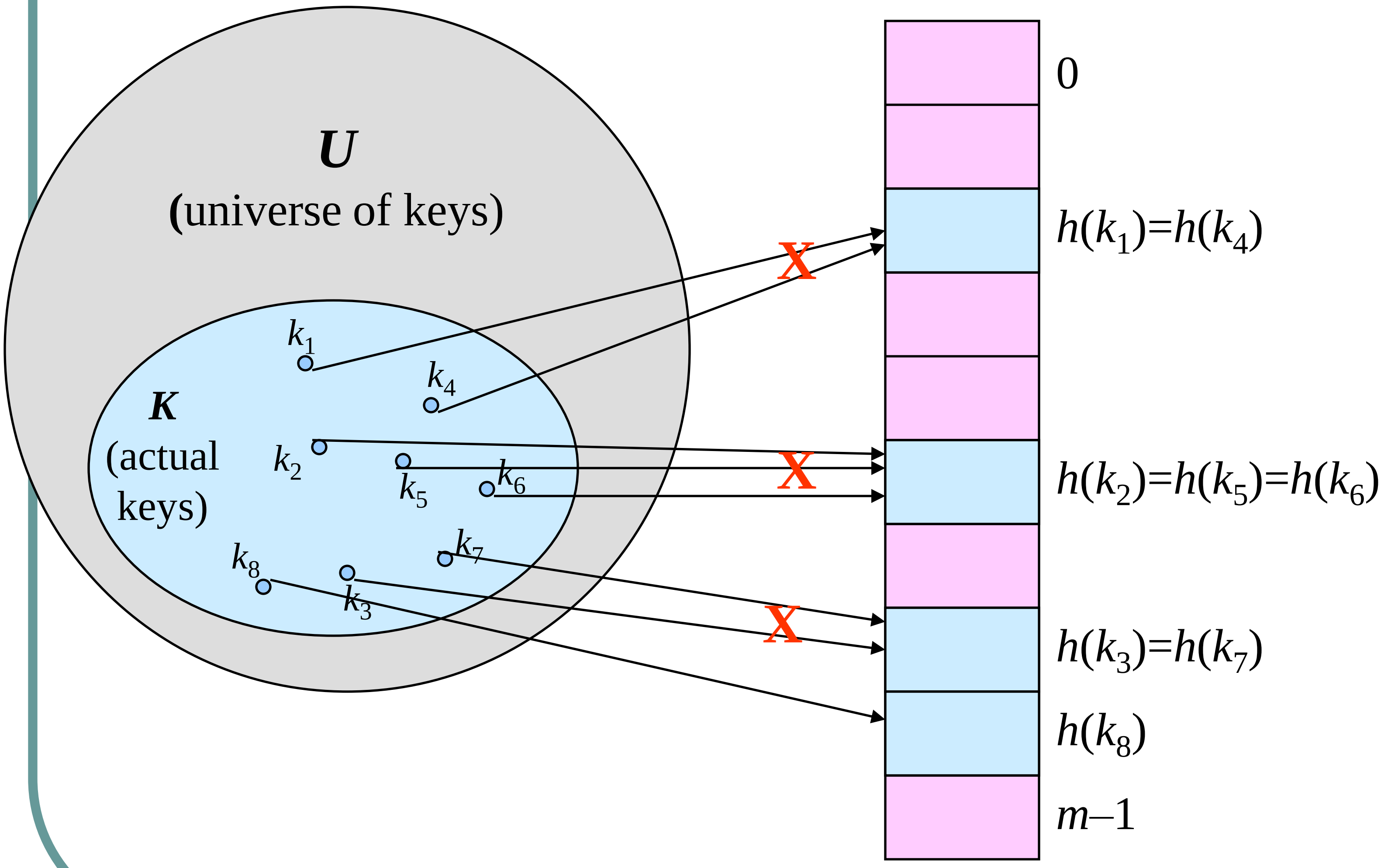
$i^2 = 1 + 3 + 5 + 7 + \ldots+(2\times i - 1), i \geq 1$

```
int inc = 1, pCount = 0;
while(pCount < i)
{
    t = (t + inc) % HTSize;
    inc = inc +2;
    pCount++;
}
```

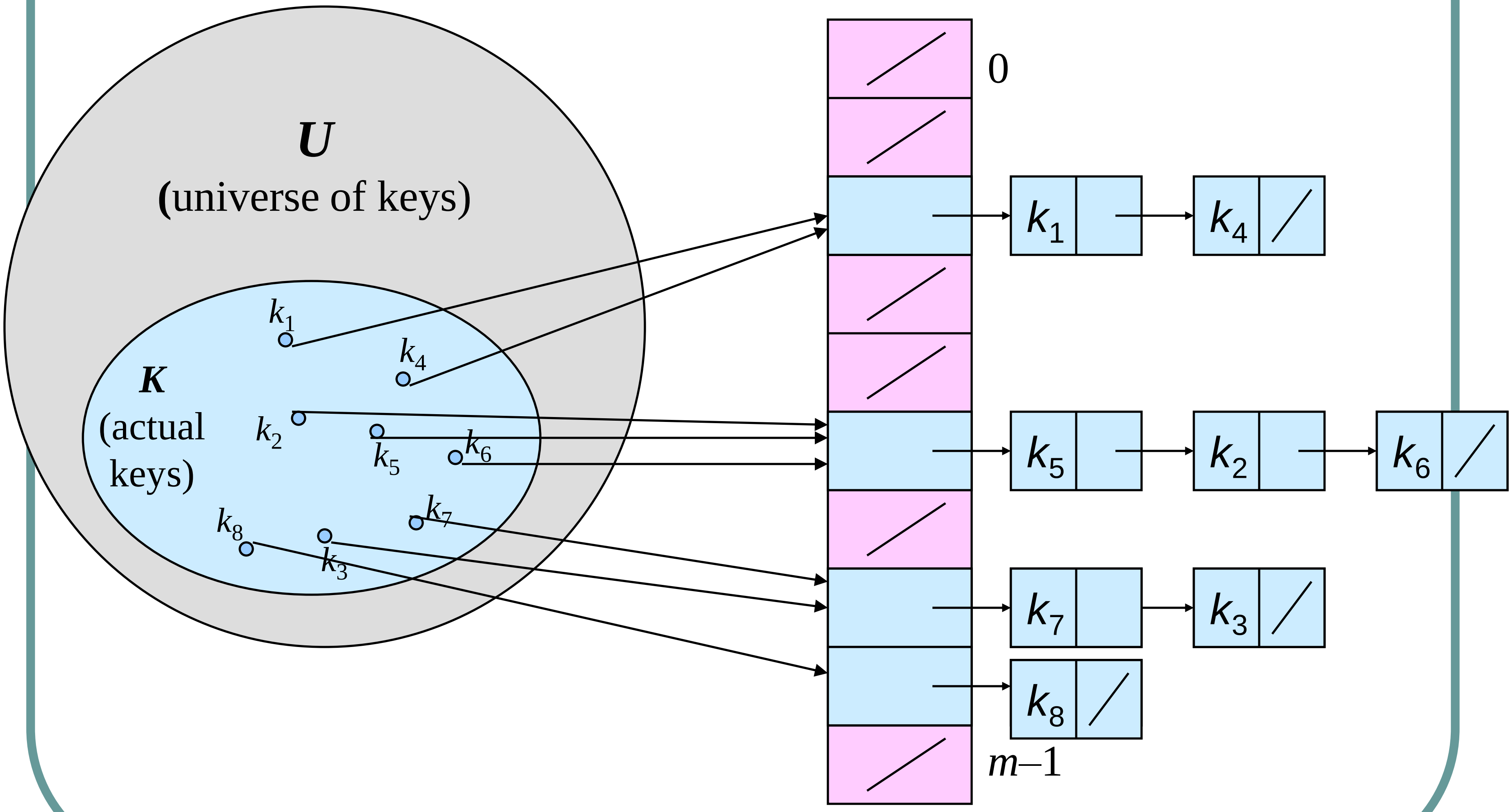
3. Collision Resolution: Chaining

- In chaining, the hash table is an array of pointers.
- For each item *x*, we first find h(*x*) = *t* where $0 \leq t \leq HT_{Size} - 1$.
- The item x is then inserted in the linked list pointed by HT[t].
- It then follows that for no identical items x_1 and x_2, if $h(x_1) = h(x_2)$, the items x_1 and x_2 are inserted in the same linked list and so collision is handled quickly.

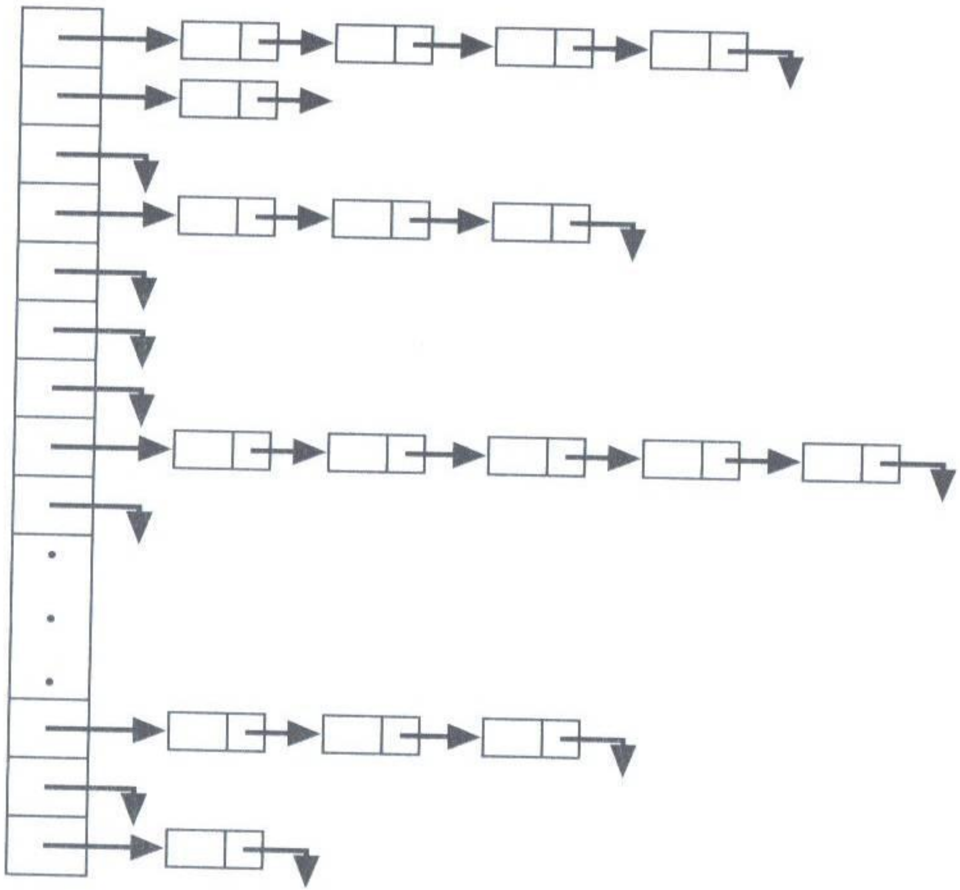
Collision Resolution by Chaining



Collision Resolution by Chaining



3. Collision Resolution: Chaining



Linked Hash table

4. Exercise

Given the following input Keys:

30, 149, 76, 107, 119, 41, 50, 91

And, hash function $h(x) = x \bmod 15$, HTSize = 15 and bucket size = 1.

A. Obtain the resulting hash table when open addressing with quadratic probing is used to resolve collisions.

B. Obtain the resulting hash table when chaining is used to resolve collisions.

4. Exercise

h(30) = 30 % 15 = 0

h(149) = 149 % 15 = 14

h(76) = 76 % 15 = 1

h(107) = 107 %15 = 2

h(119) = 119 % 15 = 14 → collision

$(14 + 1^2)$ % 15 = 15 % 15 = 0 → collision

$(14 + 2^2)$ % 15 = 18 % 15 = 3

h(41) = 41 % 15 = 11

h(50) = 50 % 15 = 5

h(91) = 91 % 15 = 1 → collision

$(1 + 1^2)$ % 15 = 2 % 15 = 2 → collision

$(1 + 2^2)$ % 15 = 5 % 15 = 5 → collision

$(1 + 3^2)$ % 15 = 10 % 15 = 10

[0]	30
[1]	76
[2]	107
[3]	119
[4]	
[5]	50
[6]	
[7]	
[8]	
[9]	
[10]	91
[11]	41
[12]	
[13]	
[14]	149

4. Exercise

h(30) = 30 % 15 = 0

h(149) = 149 % 15 = 14

h(76) = 76 % 15 = 1

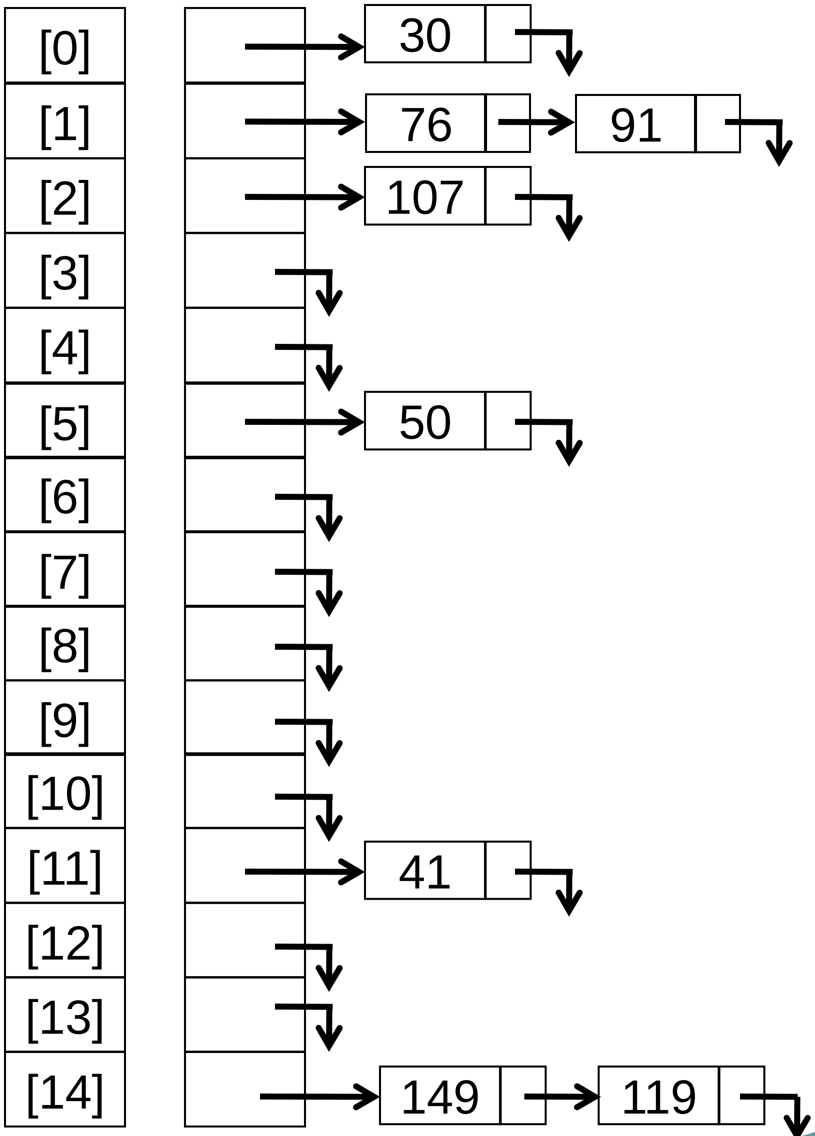
h(107) = 107 %15 = 2

h(119) = 119 % 15 = 14

h(41) = 41 % 15 = 11

h(50) = 50 % 15 = 5

h(91) = 91 % 15 = 1



4. Exercise

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