CS303 - Algorithms and Data Structures

Lecture 8

Data Structures

Professor : Mahmut Unan – UAB CS

Agenda

- Data Structures
- Stack
- Queue
- Linked List
- Hashing

Python Stack Implementation

```
In [13]: stack=[]

In [14]: stack.append("UAB")
         stack.append("CS")
         stack.append("303")
         print(stack)

         ['UAB', 'CS', '303']

In [15]: print(stack.pop())

         303

In [16]: print(stack.pop())

         CS

In [17]: print(stack.pop())

         UAB
```

Python Queue

In [22]:
```
queue=[]
```

In [23]:
```
queue.append("UAB")
queue.append("CS")
queue.append("303")
print(queue)
```

```
['UAB', 'CS', '303']
```

In [25]:
```
print(queue.pop(0))
```

```
UAB
```

In [26]:
```
print(queue.pop(0))
```

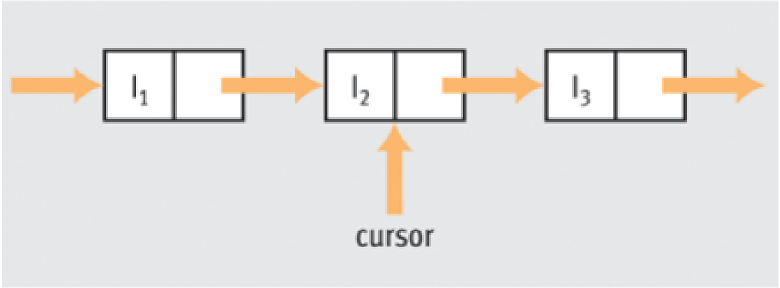
```
CS
```

In [27]:
```
print(queue.pop(0))
```

```
303
```

Linked Lists

- Linked Lists provide a simple, flexible representation for dynamic sets
  - the order in a linked list is determined by a pointer in each object
  - retrieval, insertion, deletion allowed anywhere within the structure
- **List** – ordered collection of zero or more **nodes**
- Nodes contain two **fields**
  - **Information** field (**data** field)
  - **Pointer** field (**next** field)



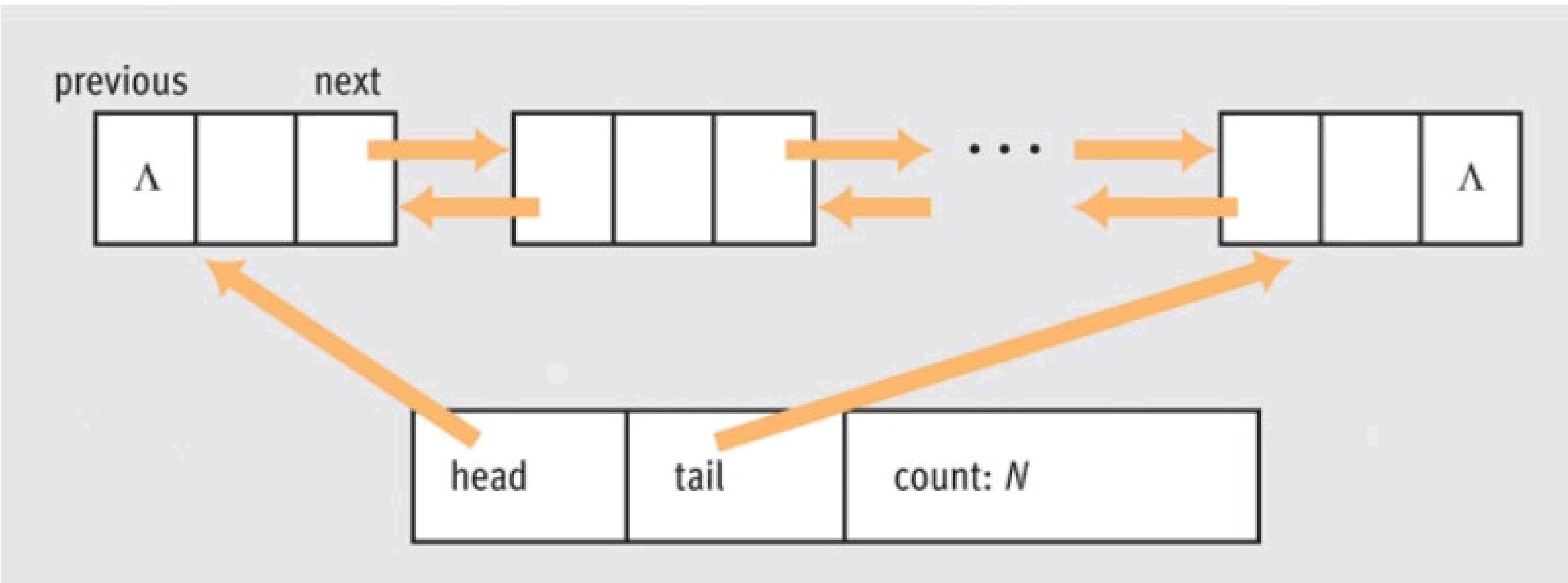
Implementations of Lists

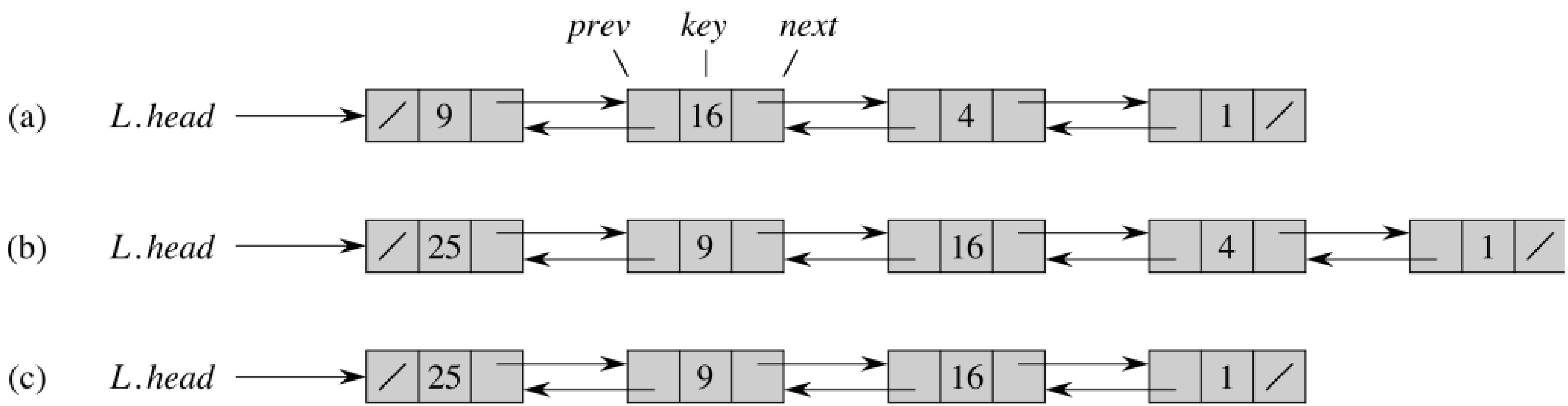
- Insert a node at the cursor:
  - Allocating space for the node
  - Assign the next field to the successor of the cursor
  - Assign the node to the next field of the cursor
  - Update total node count
- Adding a node is O(1) time complexity
- Many operations are O(1) because cursor points to the location, not using physical adjacency

Doubly Linked Lists

- **Singly linked list** – each node has a pointer to its successor
- **Doubly linked list** – nodes have a pointer to successor and predecessor
  - head and last nodes, cursor, and count
- Time somplexity to search an element O(n)
  - Doubly linked list, list insertion/deletion time complexity O(1)

Doubly linked list data structure





(b) After LIST_INSERT(L, 25)

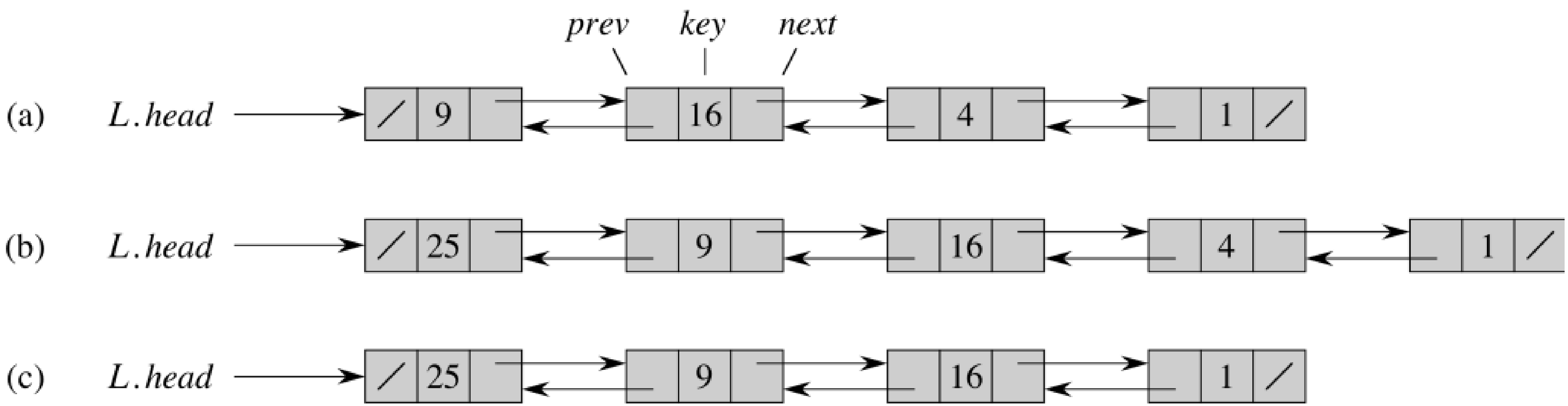
(c) After LIST_DELETE(L, 4)

LIST-INSERT(L, x)

$x.next = L.head$
if $L.head \neq$ NIL
    $L.head.prev = x$
$L.head = x$
$x.prev =$ NIL

LIST-DELETE(L, x)

if $x.prev \neq$ NIL
    $x.prev.next = x.next$
else $L.head = x.next$
if $x.next \neq$ NIL
    $x.next.prev = x.prev$



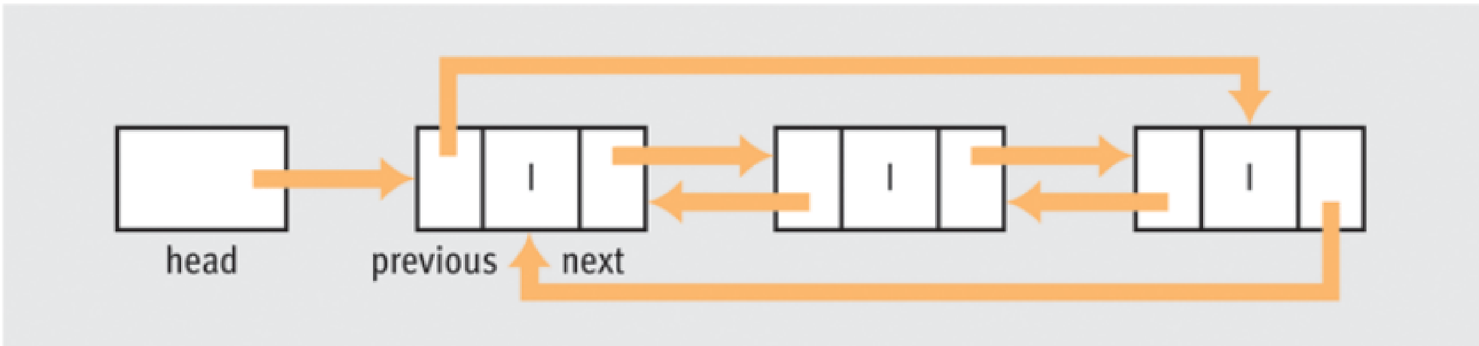
LIST-SEARCH(L, k)

```
x = L.head
while x ≠ NIL and x.key ≠ k
    x = x.next
return x
```

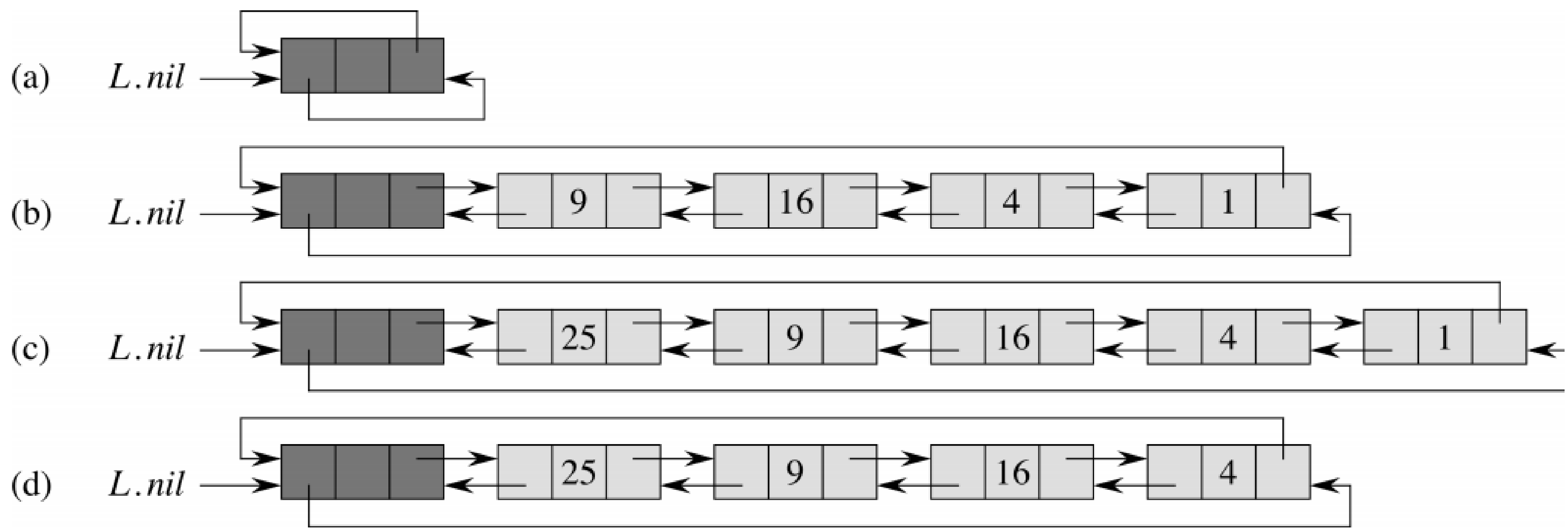
Circular Lists

- **Singly linked circular list (ring)**
  - last node next field points to the head of the list
  - No special case at ends of list
  - next() operation on last node returns first node
  - previous() operation on first node returns last node
- **Doubly linked circular list**
  - Last node successor points to the head node
  - Head node predecessor points to last node

Circular, doubly linked list



Circular, doubly linked list with a sentinel



Why Hash tables / Hashing ?

- Suppose we want to design a system for storing employee records keyed using phone numbers. And we want following queries to be performed efficiently:
  - Insert a phone number and corresponding information.
  - Search a phone number and fetch the information.
  - Delete a phone number and related information.

https://www.geeksforgeeks.org/hashing-set-1-introduction/

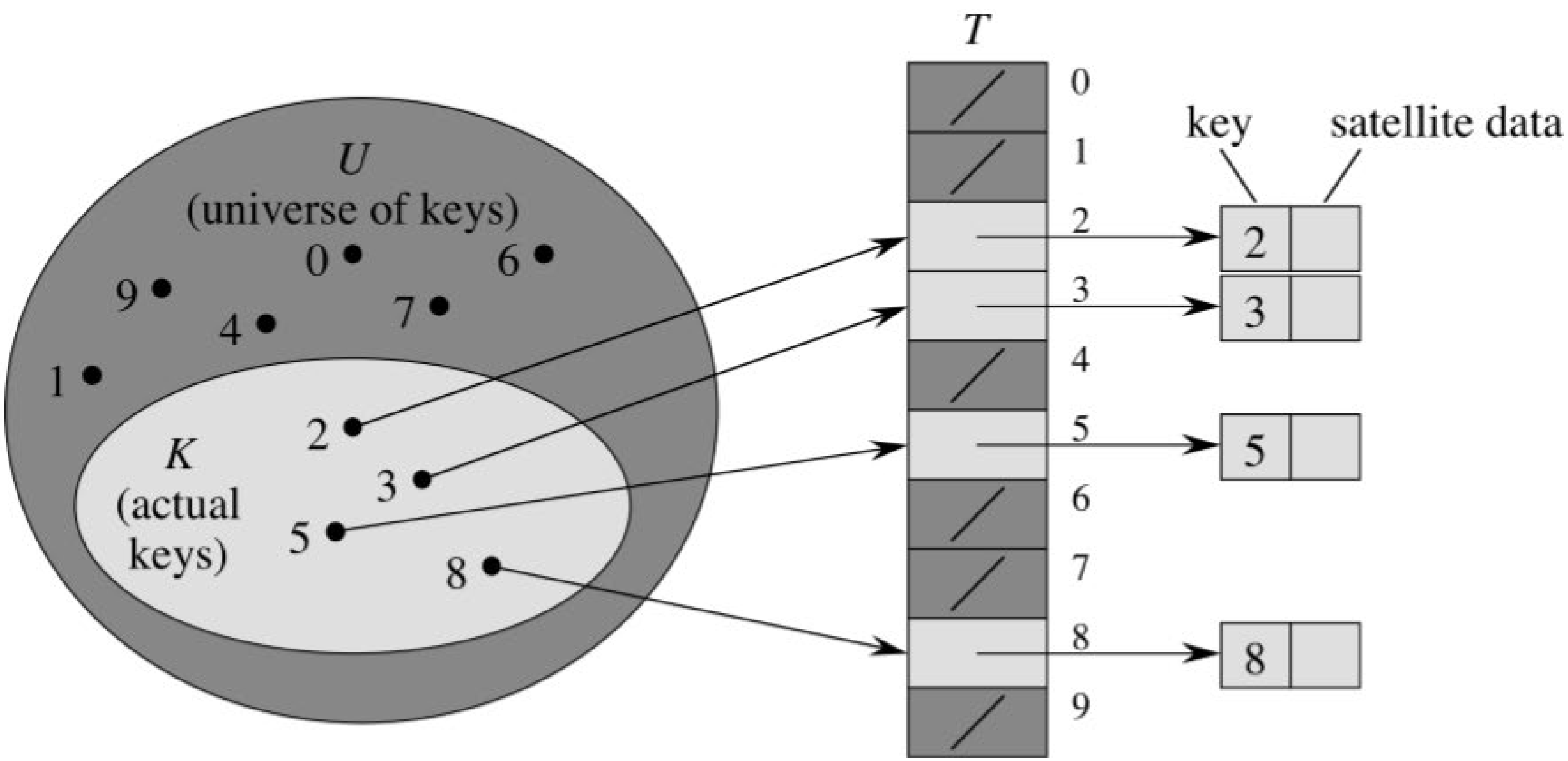
- We can think of using the following data structures to maintain information about different phone numbers.
  - Array of phone numbers and records.
  - Linked List of phone numbers and records.
  - Balanced binary search tree with phone numbers as keys.
  - Direct Access Table.

https://www.geeksforgeeks.org/hashing-set-1-introduction/

- **Direct Access Table:** here we make a big array and use phone numbers as index in the array.
- An entry in array is NIL if phone number is not present, else the array entry stores pointer to records corresponding to phone number.
- Time complexity wise this solution is the best among all, we can do all operations in O(1) time
- .....

- Need a huge storage ☹

Direct-address tables

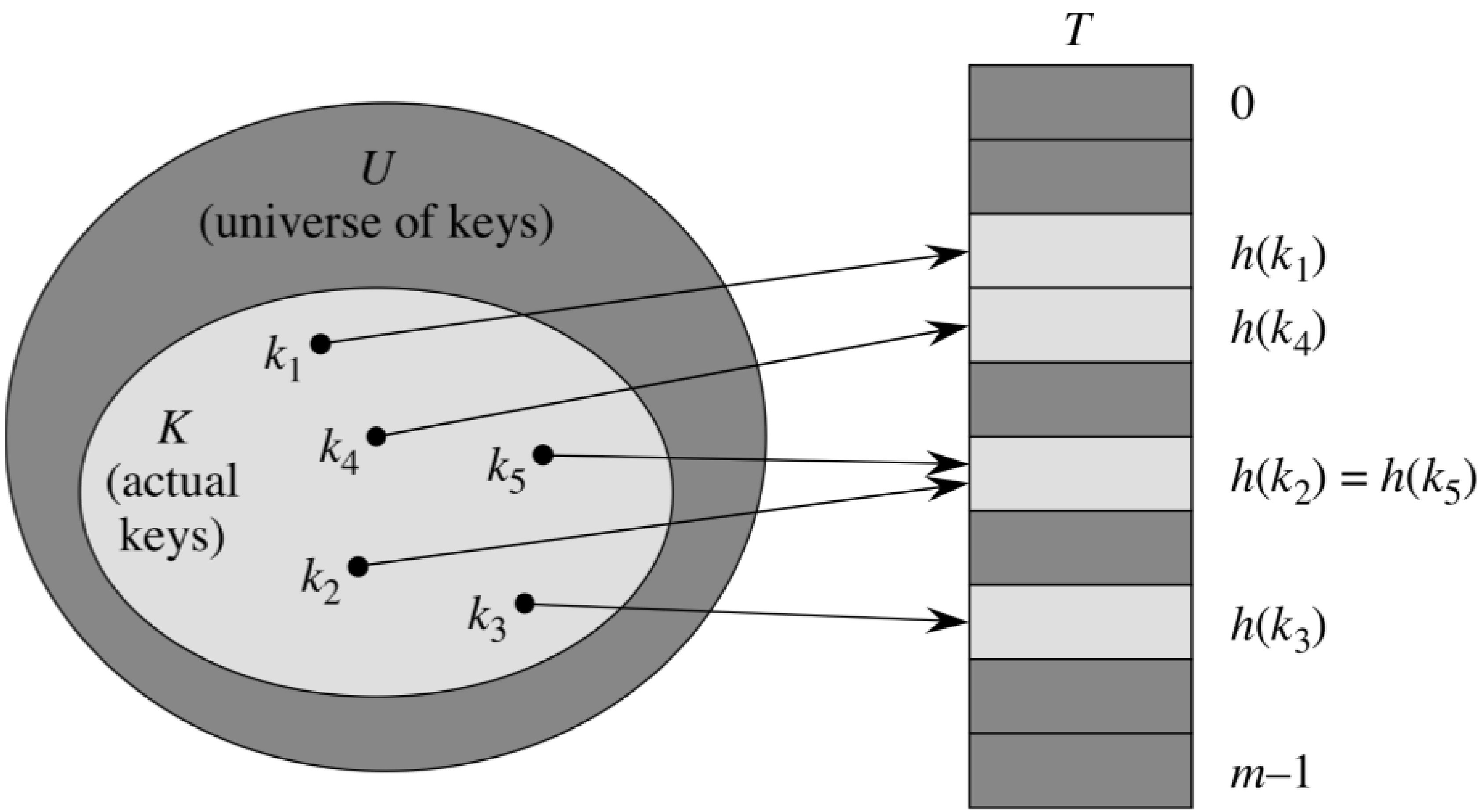


DIRECT-ADDRESS-SEARCH(T, k)
return $T[k]$

DIRECT-ADDRESS-INSERT(T, x)
$T[key[x]] = x$

DIRECT-ADDRESS-DELETE(T, x)
$T[key[x]] =$ NIL

Hash Tables



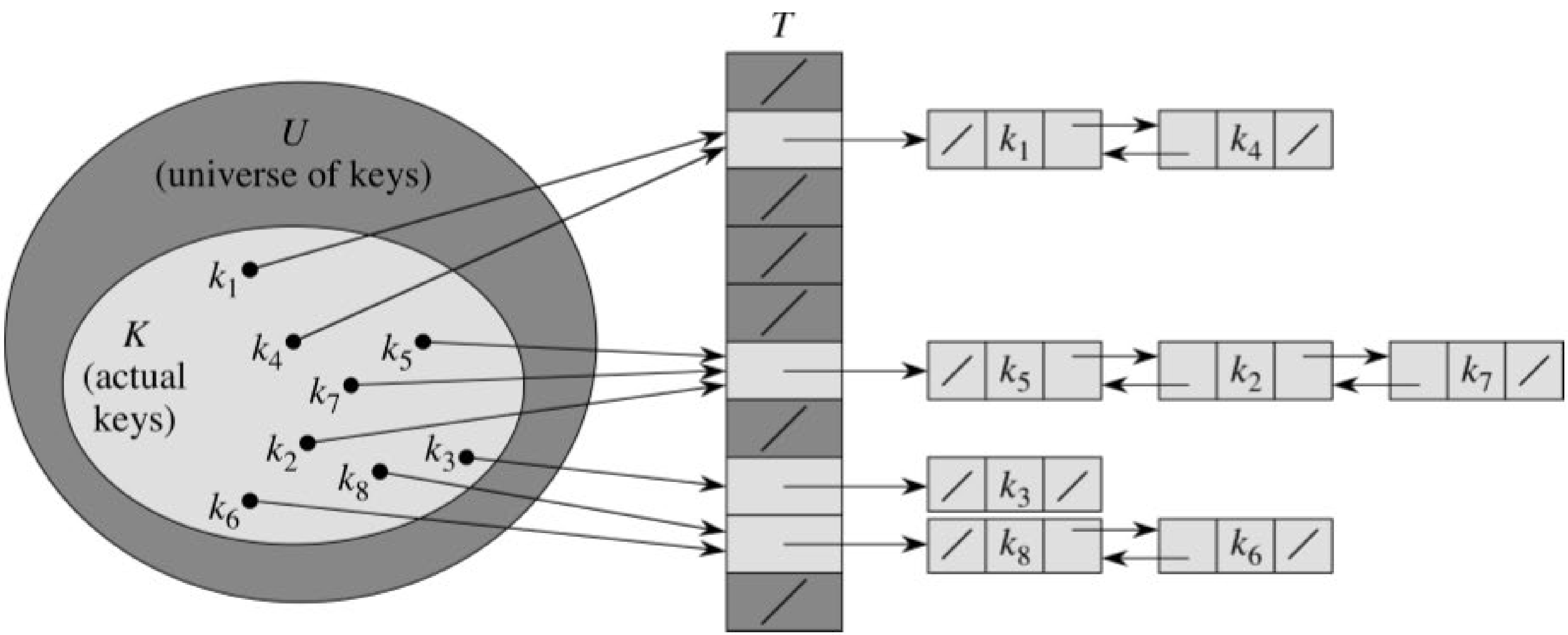
$h : U \rightarrow \{0, 1, \ldots.., m\text{-}1\}$

Problem ?

- There is one hitch: two keys may hash the to the same slot, we call this situation a **collision**
- Fortunately, we have effective techniques for resolving the conflict created by collisions. For example; chaining

- **In chaining**: we place all elements that hash to the same slot into the same linked list.

Collision resolution by chaining



CHAINED-HASH-INSERT(T, x)
insert x at the head of list $T[h(key[x])]$

CHAINED-HASH-DELETE(T, x)
delete x from the list $T[h(key[x])]$

CHAINED-HASH-SEARCH(T, k)
search for an element with key k in list $T[h(k)]$

Time Complexity

	Average	Worst Case
space	O(n)	O(n)
insert	O(1)	O(n)
lookup	O(1)	O(n)
delete	O(1)	O(n)

Hash Function

- A function that converts a given big phone number to a small practical integer value. The mapped integer value is used as an index in hash table. In simple terms, a hash function maps a big number or string to a small integer that can be used as index in hash table.
- A good hash function should have following properties
  1) Efficiently computable.
  2) Should uniformly distribute the keys (Each table position equally likely for each key)
- For example for phone numbers a bad hash function is to take first three digits. A better function is consider last three digits. Please note that this may not be the best hash function. There may be better ways.

https://www.geeksforgeeks.org/hashing-set-1-introduction/

Hash Functions

- A good hash function satisfies the assumption of a simple uniform hashing (each key is likely to hash to any of the *m* slots, independently of where any other key has hashed up)
- Heuristic approaches:
  - hashing by division
  - hashing by multiplication
- Randomization:
  - universal hashing, provides better performance on average

Hash Functions

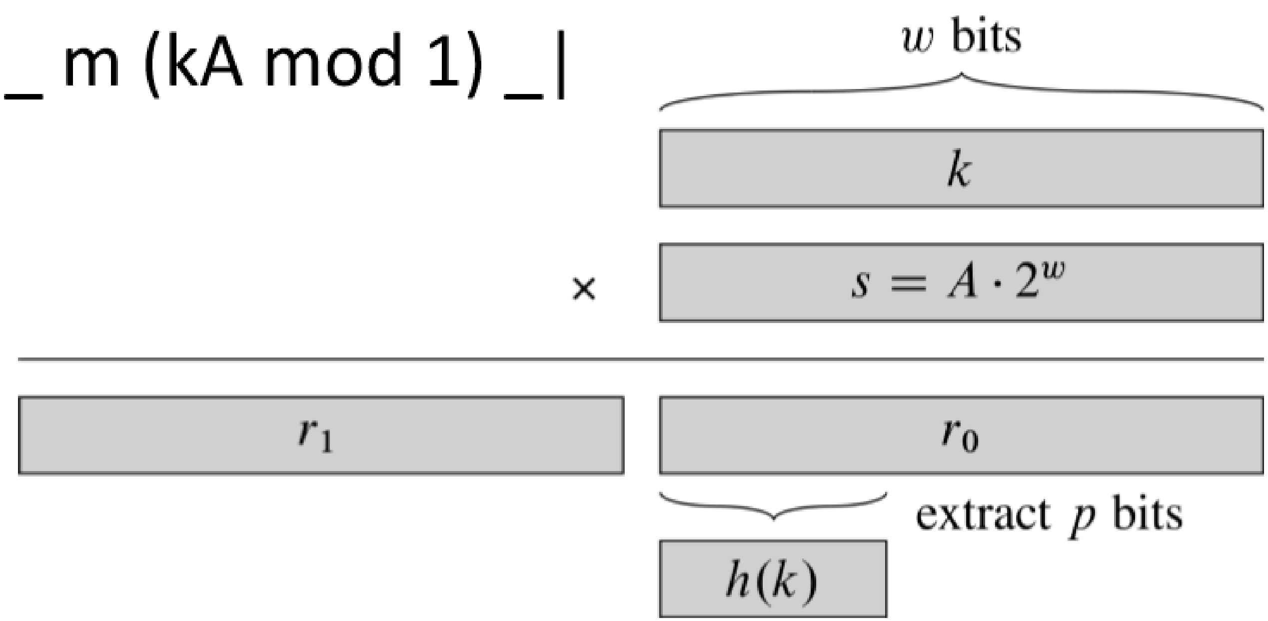
- A good hash function should satisfy the following three key requirements
  - Deterministic- equal keys should produce the same hash value
  - efficient to compute
  - Uniformly distribute the keys

Division Method

- Map a key *k* into one of *m* slots by taking the remainder of *k* divided by *m*: $h(k) = k \bmod m$
- A prime number not too close to an exact power of 2 is often a good choice for *m*
- What happens if m is a power of 2 ($m = 2^p$) ?
- Consider inserting the keys *10, 22, 31, 4, 15, 28, 17, 88, 59* into a hash table that has 11 slots. Demonstrate what happens when the keys are inserted into a hash table with collisions resolved by chaining and the hash function $h(k) = k \bmod 11$.

Multiplication Method of Hashing

- Step 1: multiply the key by a constant A (0 < A < 1) and extract the fractional part of kA
- Step 2: multiply the above value by m and take the floor of the result
- h(k) = |_ m (kA mod 1) _|



Universal hashing

- If a malicious adversary chooses the keys to be hashed by some fixed hash function, then the adversary can choose n keys that all hash to the same slot, yielding an average retrieval time of Big Theta (n).
- Any fixed hash function is vulnerable to such terrible worst-case behavior; the only effective way to improve the situation is to choose the hash function **randomly** in a way that is **independent** of the keys that are actually going to be stored.
- This approach, called **universal hashing**, can yield provably good performance on average, no matter which keys the adversary chooses.

Open Addressing

- All elements occupy the hash table itself, no lists and no elements stored outside the table
- Avoids pointers altogether, instead of following pointers, the slots to be examined are ***computed***
- To insert a key into the hash table successively examine, or ***probe***, the hash table until an empty slot is found
- The sequence of positions probed *depends upon the key being inserted*

Open Addressing

- To determine which slots to probe, the hash function is extended to include the probe number (starting from 0) as a second input
- The extended hash function is:

  $$h: U \times \{0, 1, \ldots, m-1\} \rightarrow \{0, 1, \ldots, m-1\}$$
- For every key k, the probe sequence

  $$\langle h(k,0), h(k,1), \ldots, h(k, m-1) \rangle$$

  will be a permutation of $\langle 0, 1, \ldots, m-1 \rangle$
- Commonly used techniques to compute the probe sequences:
  - linear probing
  - quadratic probing
  - double hashing

Linear Probing

- Given an auxiliary hash function, $h' : U \rightarrow \{0, 1, \ldots, m-1\}$, linear probing uses the hash function $h(k,i) = (h'(k) + i) \bmod m$, for i = 0, 1, …, m-1
- Given key k, linear probing works as follows:
  - first probe $T[h'(k)]$, *i.e.*, the slot given by the auxiliary hash function
  - next probe slot $T[h'(k)+1]$, and so on up to slot $T[m-1]$
  - then wrap around to slots $T[0]$, $T[1]$, … until we finally probe slot $T[h'(k)-1]$
- Disadvantage: primary clustering – long runs of occupied slots tend to get longer and the average search time increases

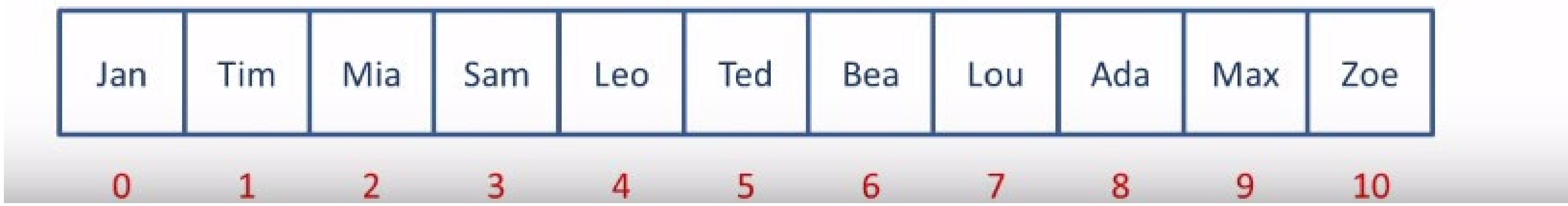
Quadratic Probing

- Uses a hash function of the form
  $$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m,$$
  where h' is an auxiliary hash function,
  c_1 and c_2 are positive auxiliary constants, and
  $i=0,1,\ldots,m-1$
- The initial position probed is $T[h'(k)]$, later positions probed are offset by amounts that depend in a quadratic manner on the probe number i
- As in linear probing, the initial probe determines the entire sequence, and so only m distinct probe sequences are used

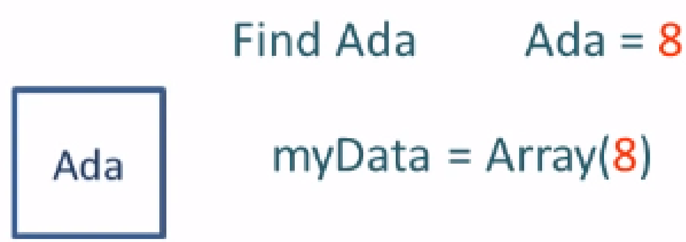
Double hashing

- Uses a hash function of the form
  $$h(k,i) = (h_1(k) + ih_2(k)) \bmod m,$$
  where h_1 and h_2 are auxiliary hash functions
- Initial probe goes to position $T[h_1(k)]$ (since $i=0$)
- Successive probe positions are offset from previous positions by the amount $h_2(k) \bmod m$
- Unlike linear or quadratic probing, the probe sequence here depends in two ways upon the key k, since the initial probe position, the offset, or both, may vary

Exercise



Search Ada ?



https://www.youtube.com/watch?v=KyUTuwz_b7Q&t=217s

Mia	M	77	i	105	a	97	279	4
Tim	T	84	i	105	m	109	298	1
Bea	B	66	e	101	a	97	264	0
Zoe	Z	90	o	111	e	101	302	5
Jan	J	74	a	97	n	110	281	6
Ada	A	65	d	100	a	97	262	9
Leo	L	76	e	101	o	111	288	2
Sam	S	83	a	97	m	109	289	3
Lou	L	76	o	111	u	117	304	7
Max	M	77	a	97	x	120	294	8
Ted	T	84	e	101	d	100	285	10

Bea	Tim	Leo	Sam	Mia	Zoe	Jan	Lou	Max	Ada	Ted
0	1	2	3	4	5	6	7	8	9	10

Index number = *sum ASCII codes* Mod *size of array*

Find Ada   Ada = (65 + 100 + 97) = 262

Find Ada   262 Mod 11 = 9

myData = Array(9)

Bea	Tim	Leo	Sam	Mia	Zoe	Jan	Lou	Max	Ada	Ted
0	1	2	3	4	5	6	7	8	9	10

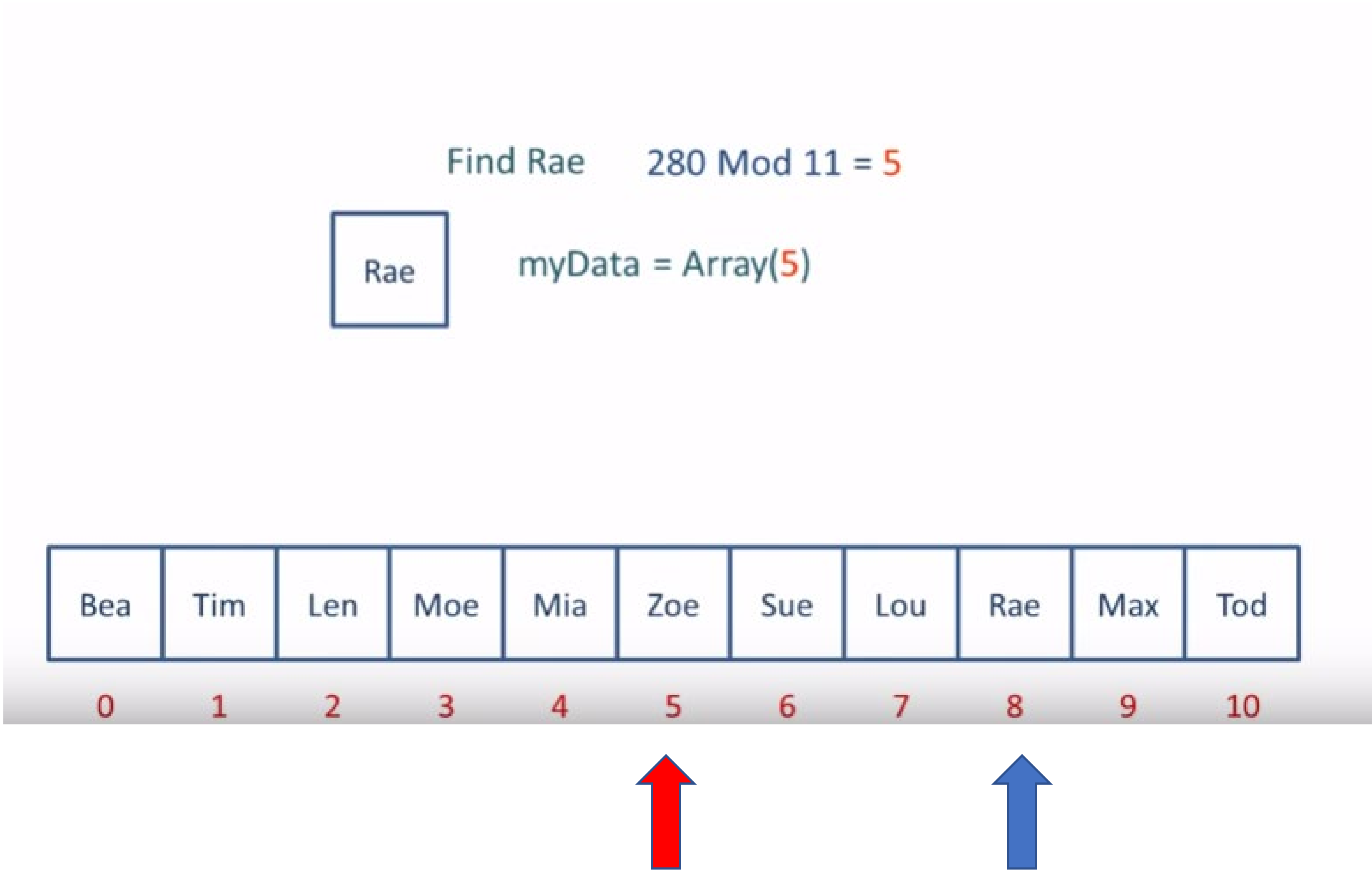
Hash Tables are used to store objects

Bea	Tim	Leo	Sam	Mia	Zoe	Jan	Lou	Max	Ada	Ted
27/01/1941	08/06/1955	31/12/1945	27/04/1791	20/02/1986	19/06/1978	13/02/1956	27/12/1822	23/04/1858	10/12/1815	17/06/1937
English	English	American	American	Russian	American	Polish	French	German	English	American
Astronomer	Inventor	Mathematician	Inventor	Space Station	Actress	Logician	Biologist	Physicist	Mathematician	Philosopher
0	1	2	3	4	5	6	7	8	9	10

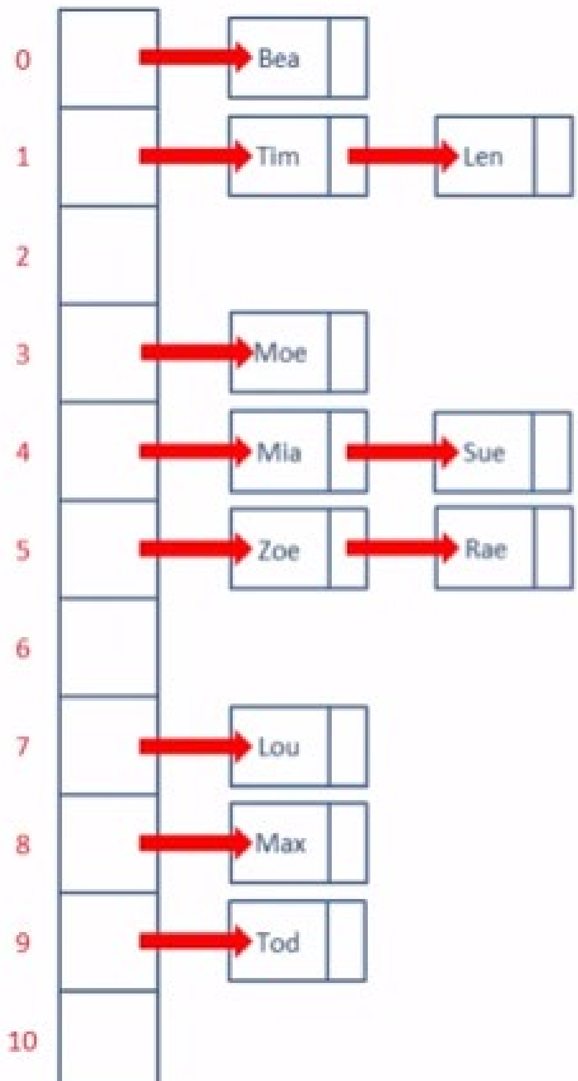
Open Addressing – Linear Probing

Mia	M	77	i	105	a	97		279	4
Tim	T	84	i	105	m	109		298	1
Bea	B	66	e	101	a	97		264	0
Zoe	Z	90	o	111	e	101		302	5
Sue	S	83	u	117	e	101		301	4
Len	L	76	e	101	n	110		287	1
Moe	M	77	o	111	e	101		289	3
Lou	L	76	o	111	u	117		304	7
Rae	R	82	a	97	e	101		280	5
Max	M	77	a	97	x	120		294	8
Tod	T	84	o	111	d	100		295	9

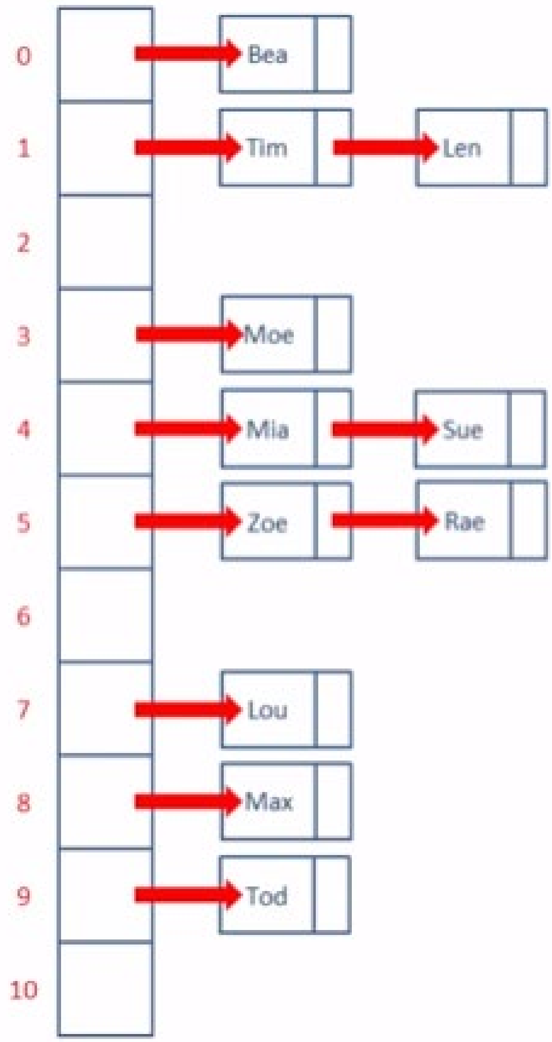
Bea	Tim	Len	Moe	Mia	Zoe	Sue	Lou	Rae	Max	Tod
0	1	2	3	4	5	6	7	8	9	10


Find Rae
280 Mod 11 = 5
Rae
myData = Array(5)
Bea
Tim
Len
Moe
Mia
Zoe
Sue
Lou
Rae
Max
Tod
0
1
2
3
4
5
6
7
8
9
10

Chaining

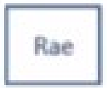


Mia	M	77	i	105	a	97	279	4
Tim	T	84	i	105	m	109	298	1
Bea	B	66	e	101	a	97	264	0
Zoe	Z	90	o	111	e	101	302	5
Sue	S	83	u	117	e	101	301	4
Len	L	76	e	101	n	110	287	1
Moe	M	77	o	111	e	101	289	3
Lou	L	76	o	111	u	117	304	7
Rae	R	82	a	97	e	101	280	5
Max	M	77	a	97	x	120	294	8
Tod	T	84	o	111	d	100	295	9



Find Rae 280 Mod 11 = 5

myData = Array(5)



Exercise (Textbook 11.4-1)

- Consider inserting the keys 10, 22, 31, 4, 15, 28, 17, 88, 59 into a hash table of length $m = 11$ using open addressing with the primary hash function $h'(k) = k \bmod m$.
- Illustrate the result of inserting these keys using linear probing, using quadratic probing with $c1 = 1$ and $c2 = 3$, and using double hashing with $h2(k) = 1 + (k \bmod (m - 1))$.

Linear Probing

- $h(k, i) = (k + i) \bmod 11.$

index	linear probing
0	22
1	88
2	
3	
4	4
5	15
6	28
7	17
8	59
9	31
10	10

10 $k = 10, i = 0, h(10, 0) = (10 + 0) \bmod 11 = 10$
22 $k = 22, i = 0, h(22, 0) = (22 + 0) \bmod 11 = 0$
31 $k = 31, i = 0, h(31, 0) = (31 + 0) \bmod 11 = 9$
4 $k = 4, i = 0, h(4, 0) = (4 + 0) \bmod 11 = 4$

15 $k = 15, i = 0, h(15, 0) = (15 + 0) \bmod 11 = 4$, collision!
$k = 15, i = 1, h(15, 1) = (15 + 1) \bmod 11 = 5$

28 $k = 28, i = 0, h(28, 0) = (28 + 0) \bmod 11 = 6$

17 $k = 17, i = 0, h(17, 0) = (17 + 0) \bmod 11 = 6$, collision!
$k = 17, i = 1, h(17, 1) = (17 + 1) \bmod 11 = 7$

88 $k = 88, i = 0, h(88, 0) = (88 + 0) \bmod 11 = 0$, collision!
$k = 88, i = 1, h(88, 1) = (88 + 1) \bmod 11 = 1$

59 $k = 59, i = 0, h(59, 0) = (59 + 0) \bmod 11 = 4$, collision!
$k = 59, i = 1, h(59, 1) = (59 + 1) \bmod 11 = 5$, collision!
$k = 59, i = 2, h(59, 2) = (59 + 2) \bmod 11 = 6$, collision!
$k = 59, i = 3, h(59, 3) = (59 + 3) \bmod 11 = 7$, collision!
$k = 59, i = 4, h(59, 4) = (59 + 4) \bmod 11 = 8$

Quadratic Probing

h(k, i) = (k + i + 3i^2) mod 11.

index	quadratic probing
0	22
1	
2	88
3	17
4	4
5	
6	28
7	59
8	15
9	31
10	10

10 k = 10, i = 0, h(10, 0) = (10 + 0 + 0) mod 11 = 10
22 k = 22, i = 0, h(22, 0) = (22 + 0 + 0) mod 11 = 0
31 k = 31, i = 0, h(31, 0) = (31 + 0 + 0) mod 11 = 9
4 k = 4, i = 0, h(4, 0) = (4 + 0 + 0) mod 11 = 4
15 k = 15, i = 0, h(15, 0) = (15 + 0 + 0) mod 11 = 4, collision!
k = 15, i = 1, h(15, 1) = (15 + 1 + 3) mod 11 = 8
28 k = 28, i = 0, h(28, 0) = (28 + 0 + 0) mod 11 = 6
17 k = 17, i = 0, h(17, 0) = (17 + 0 + 0) mod 11 = 6, collision!
k = 17, i = 1, h(17, 1) = (17 + 1 + 3) mod 11 = 10, collision!
k = 17, i = 2, h(17, 2) = (17 + 2 + 12) mod 11 = 9, collision!
k = 17, i = 3, h(17, 3) = (17 + 3 + 27) mod 11 = 3
88 k = 88, i = 0, h(88, 0) = (88 + 0 + 0) mod 11 = 0, collision!
k = 88, i = 1, h(88, 1) = (88 + 1 + 3) mod 11 = 4, collision!
k = 88, i = 2, h(88, 2) = (88 + 2 + 12) mod 11 = 3, collision!
k = 88, i = 3, h(88, 3) = (88 + 3 + 27) mod 11 = 8, collision!
k = 88, i = 4, h(88, 4) = (88 + 4 + 48) mod 11 = 8, collision!
k = 88, i = 5, h(88, 5) = (88 + 5 + 75) mod 11 = 3, collision!
k = 88, i = 6, h(88, 6) = (88 + 6 + 108) mod 11 = 4, collision!
k = 88, i = 7, h(88, 7) = (88 + 7 + 147) mod 11 = 0, collision!
k = 88, i = 8, h(88, 8) = (88 + 8 + 192) mod 11 = 2
59 k = 59, i = 0, h(59, 0) = (59 + 0 + 0) mod 11 = 4, collision!
k = 59, i = 1, h(59, 1) = (59 + 1 + 3) mod 11 = 8, collision!
k = 59, i = 2, h(59, 2) = (59 + 2 + 12) mod 11 = 7

Double hashing

h(k, i) = (k + i(1 + (k mod 10))) mod 11.

index	double hashing
0	22
1	
2	59
3	17
4	4
5	15
6	28
7	88
8	
9	31
10	10

10 k = 10, i = 0, h(10, 0) = 10
22 k = 22, i = 0, h(22, 0) = 0
31 k = 31, i = 0, h(31, 0) = 9
4 k = 4, i = 0, h(4, 0) = 4

15 k = 15, i = 0, h(15, 0) = 4, collision!
k = 15, i = 1, h(15, 1) = 10, collision!
k = 15, i = 2, h(15, 2) = 5

28 k = 28, i = 0, h(28, 0) = 6

17 k = 17, i = 0, h(17, 0) = 6, collision!
k = 17, i = 1, h(17, 1) = 3

88 k = 88, i = 0, h(88, 0) = 0, collision!
k = 88, i = 1, h(88, 1) = 9, collision!
k = 88, i = 2, h(88, 2) = 7

59 k = 59, i = 0, h(59, 0) = 4, collision!
k = 59, i = 1, h(59, 1) = 3, collision!
k = 59, i = 2, h(59, 2) = 2