HASH TABLE

DICTIONARY

An example : Python 3

```
>>> numNames={1:"One", 2: "Two", 3:"Three"}
>>> numNames.get(2)
'Two'
>>> del numNames[2]
>>> numNames
{1: 'One', 3: 'Three'}
>>> numNames[2] = 'Two'
>>> numNames
{1: 'One', 3: 'Three', 2: 'Two'}
```

```
>>> romanNums = {'I':1, 'II':2, 'III':3, 'IV':4, 'V':5}
>>> romanNums.get('IV')
4
>>> del romanNums['IV']
>>> romanNums
{'I': 1, 'II': 2, 'III': 3, 'V': 5}
>>> romanNums['IV'] = 4
>>> romanNums
{'I': 1, 'II': 2, 'III': 3, 'V': 5, 'IV': 4}
>>> romanNums.get('X')
>>> romanNums.get('X') == None
True
```

Dictionary of n keys

Data Structure	Insert	Search	Delete
Unsorted linked list	O(1)	O(n)	O(n)
Unsorted array	O(1)	O(n)	O(n)
Sorted linked list	O(n)	O(n)	O(n)
Sorted array	O(n)	O(lg n)	O(n)
Balanced search tree	O(lg n)	O(lg n)	O(lg n)
Hash Table	O(1)	O(1)	O(1)

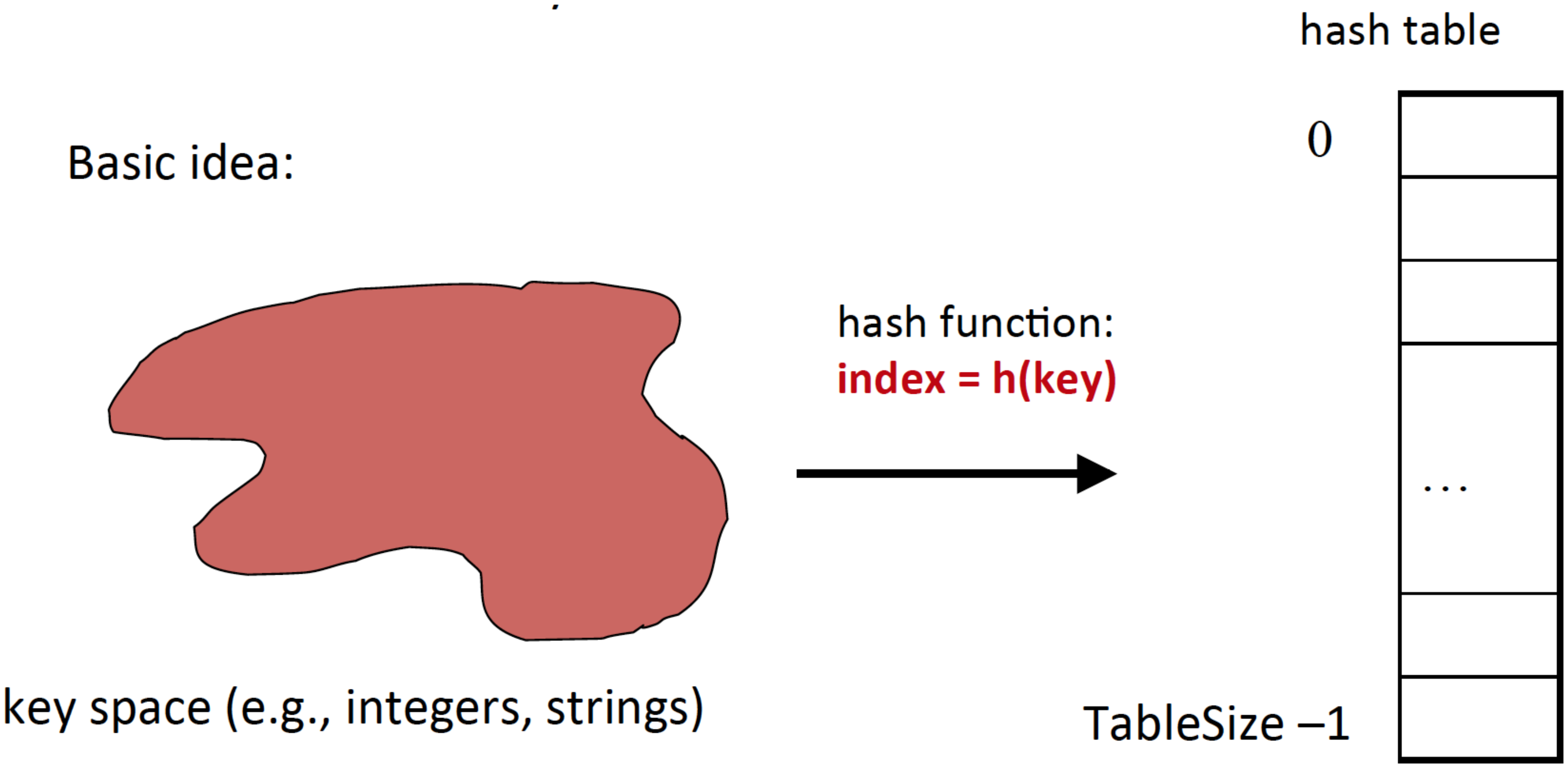
Direct-Address Table

- Counting the frequency of integers in a text file.
- Integer value is guaranteed to be in range [0,100]
- Ex. 5, 6, 3, 99, 5, 0, 0, 1, 6
- Have table size proportional to number of keys while maintaining same average access speed?

Index (key)	count
0	2
1	1
2	0
3	1
4	0
5	2
6	2
:	:
99	1
100	0

Hash Table

Basic idea:



Hash Function

Let h(x) = x % 15. Then,

- if x = 25 129 35 2501 47 36
- h(x) = 10 9 5 11 2 6

Storing the keys in the array is straightforward:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
_	_	47	_	_	35	36	_	_	129	25	2501	_	_	_

Thus, delete and search can be done in O(1), and also insert, except…

Hash Function

What happens when trying to insert: x = 65 ?

x = 65

h(x) = 5

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
_	_	47	_	_	35 65(?)	36	_	_	129	25	2501	_	_	_

This is called a collision.

Hash Table issues

Size

Hash function

Handling collision

- Separate chaining
- Open addressing
  - Linear probing
  - Quadratic probing
  - Double hashing

Hash Table Size

A good general "rule of thumb":

- The hash table size should be about 1.3 times the maximum number of keys that will actually be in the table
- Size of hash table should be a prime number

Resize when needed

- A recommendation is to keep the ration between keys and table size in range [α/4, α]
- α is the ratio between max number of keys and table size such that the average running time is acceptable as O(1)

Designing Hash Function

- Often, h(k) = k % m ; m is the table size
- Options for string key:

  Let s = $s_0, s_1, s_2, \ldots\ , s_{L-1}$
  - h(s) = $ascii(s_0)\ \%\ m$
  - h(s) = $(\sum_i ascii(s_i))\ \%\ m$
  - h(s) = $(\sum_i 37^i\ ascii(s_i))\ \%\ m$

Handling Collision : Separate Chaining

Let each array element be the head of a chain:

Where would you store: 29, 16, 14, 99, 127 ?

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	↓	↓			↓	↓	↓		↓	↓	↓			↓
	16	47			65	36	127		99	25	2501			14
					↓				↓					↓
					35				129					29

New keys go at the front of the relevant chain.

Handling Collision: Open Addressing

If hash table is not full,

- Repeat until an empty slot is found:
  - Attempt to store the key in the next choice

On i^{th} attempt:

- Linear Probing : target index = (h(k) + i) % m
- Quadratic Probing : target index = (h(k) + i^2) % m
  - in order to avoid consecutive occupations of slot
- Double Hashing: target index = (h(k) + i*g(k)) % m
  - Typically, g(k) = R − (k % R) where R is a prime number < m

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Open Addressing: Delete

- Assume linear probing.
- H=KEY MOD 10
- Insert 47, 57, 68, 18, 67
- Search 68
- Search 10
- Delete 47
- Search 57

Deletion-Aware Algorithms

- Insert

• Cell empty or deleted	insert at H, *cell* = *active*
• Cell active	H = (H + 1) % Table_Size

- Search

• cell empty	NOT found
• cell deleted	H = (H + 1) mod Table_Size
• cell active	if key == key in cell -> FOUND else H = (H + 1) % Table_Size

- Delete

• cell active; key != key in cell	H = (H + 1) % Table_Size
• cell active; key == key in cell	DELETE; *cell=deleted*
• cell deleted	H = (H + 1) % Table_Size
• cell empty	NOT found