Lecture 4: Hash Tables

김수경
이화여자대학교 인공지능융합전공 소속

저작권 안내

(주)업스테이지가 제공하는 모든 교육 콘텐츠의 지식재산권은
운영 주체인 (주)업스테이지 또는 해당 저작물의 적법한 관리자에게 귀속되어 있습니다.

콘텐츠 일부 또는 전부를 복사, 복제, 판매, 재판매 공개, 공유 등을 할 수 없습니다.

유출될 경우 지식재산권 침해에 대한 책임을 부담할 수 있습니다.

유출에 해당하여 금지되는 행위의 예시는 다음과 같습니다.

- 콘텐츠를 재가공하여 온/오프라인으로 공개하는 행위
- 콘텐츠의 일부 또는 전부를 이용하여 인쇄물을 만드는 행위
- 콘텐츠의 전부 또는 일부를 녹취 또는 녹화하거나 녹취록을 작성하는 행위
- 콘텐츠의 전부 또는 일부를 스크린 캡쳐하거나 카메라로 촬영하는 행위
- 지인을 포함한 제3자에게 콘텐츠의 일부 또는 전부를 공유하는 행위
- 다른 정보와 결합하여 Upstage Education의 콘텐츠임을 알아볼 수 있는 저작물을 작성, 공개하는 행위
- 제공된 데이터의 일부 혹은 전부를 Upstage Education 프로젝트/실습 수행 이외의 목적으로 사용하는 행위

Data Structures So Far

	Insertion	Retrieval	Deletion	Restriction
Sorted Array	O(*M*)	O(log *M*)	O(*M*)	
Sorted Linked list	O(*M*)	O(*M*)	O(*M*)	
Stack	O(1)	O(1)	O(1)	LIFO
Queue	O(1)	O(1)	O(1)	FIFO
Hash Table	O(1)	O(1)	O(1)	

Needed for applications that need radically fast operations:

- 911 emergency calls and locating caller's address
- Airline information system
- Web search

Data Indexed Arrays

Data Indexed Arrays

- A regular list: [2, 5, 9, 10]
  - In this setting, we take O(*N*) to search a particular value.

Index	[0]	[1]	[2]	[3]
Value	2	5	9	10

- What if we represent the data in a **data-indexed array**?

Index	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
Value	F	F	**T**	F	F	**T**	F	F	F	**T**	**T**

Data Indexed Arrays

- A data-indexed array has **all possible data as its indices**!
  - Initially, all values of the array are **False** (*i.e.*, `di_array[x] = False` for all `x` in `di_array`), meaning the array is empty.
  - When we **insert** some value, the corresponding index becomes **True**.
  - Here, we assume that no duplicate keys are allowed.

```
di_array = DataIndexedArray()
di_array.insert(3)
di_array.insert(6)
di_array.delete(3)
```

Index	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
Value	F	F	F	T	F	F	T	F	F	F	F

Data Indexed Arrays

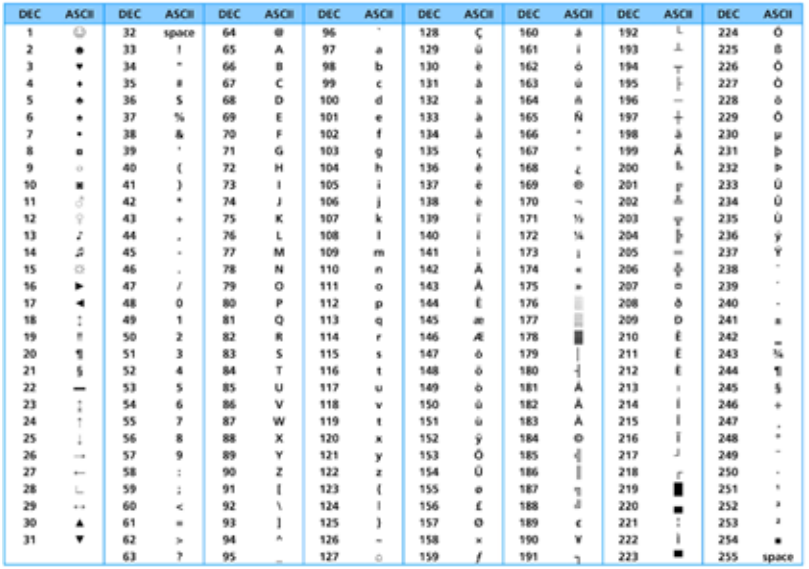
- Why is this good?
  - We can insert, delete, and retrieve values in **O(1)**!
- Any problem then?
  - We can't deal with values larger than 10 (the array size).
  - In other words, we need to use **very large memory space** if we'd like to deal with large number, even though the data size itself is small.
  - It is **not trivial how to deal with non-integer values**, like string or floating point numbers.

```
di_array.insert(21057381)
```

```
di_array.insert("ewha")
```

Strings in Data Indexed Arrays

- Recall that each character can be represented as an ASCII code value (0~255).
  - “ewha”:
    - e: 101
    - w: 87
    - h : 104
    - a: 97
  - “ewha”$_{256}$: $(101 \times 256^3) + (87 \times 256^2) + (104 \times 256) + 97$
- In this way, a string can be represented as a **unique integer**.



DEC	ASCII	DEC	ASCII	DEC	ASCII	DEC	ASCII	DEC	ASCII	DEC	ASCII	DEC	ASCII	DEC	ASCII
1	☺	32	space	64	@	96	`	128	Ç	160	á	192	└	224	Ó
2	☻	33	!	65	A	97	a	129	ü	161	í	193	┴	225	ß
3	♥	34	"	66	B	98	b	130	é	162	ó	194	┬	226	Ô
4	♦	35	#	67	C	99	c	131	â	163	ú	195	├	227	Ò
5	♣	36	$	68	D	100	d	132	ä	164	ñ	196	─	228	õ
6	♠	37	%	69	E	101	e	133	à	165	Ñ	197	┼	229	Õ
7	•	38	&	70	F	102	f	134	å	166	ª	198	ã	230	µ
8	◘	39	'	71	G	103	g	135	ç	167	º	199	Ã	231	þ
9	○	40	(	72	H	104	h	136	ê	168	¿	200	╚	232	Þ
10	◙	41	)	73	I	105	i	137	ë	169	®	201	╔	233	Ú
11	♂	42	*	74	J	106	j	138	è	170	¬	202	╩	234	Û
12	♀	43	+	75	K	107	k	139	ï	171	½	203	╦	235	Ù
13	♪	44	,	76	L	108	l	140	î	172	¼	204	╠	236	ý
14	♫	45	-	77	M	109	m	141	ì	173	¡	205	═	237	Ý
15	☼	46	.	78	N	110	n	142	Ä	174	«	206	╬	238	¯
16	►	47	/	79	O	111	o	143	Å	175	»	207	¤	239	´
17	◄	48	0	80	P	112	p	144	É	176	░	208	ð	240	
18	↕	49	1	81	Q	113	q	145	æ	177	▒	209	Ð	241	±
19	‼	50	2	82	R	114	r	146	Æ	178	▓	210	Ê	242	‗
20	¶	51	3	83	S	115	s	147	ô	179	│	211	Ë	243	¾
21	§	52	4	84	T	116	t	148	ö	180	┤	212	È	244	¶
22	▬	53	5	85	U	117	u	149	ò	181	Á	213	ı	245	§
23	↨	54	6	86	V	118	v	150	û	182	Â	214	Í	246	÷
24	↑	55	7	87	W	119	w	151	ù	183	À	215	Î	247	¸
25	↓	56	8	88	X	120	x	152	ÿ	184	©	216	Ï	248	°
26	→	57	9	89	Y	121	y	153	Ö	185	╣	217	┘	249	¨
27	←	58	:	90	Z	122	z	154	Ü	186	║	218	┌	250	·
28	∟	59	;	91	[	123	{	155	ø	187	╗	219	█	251	¹
29	↔	60	<	92	\	124	|	156	£	188	╝	220	▄	252	³
30	▲	61	=	93	]	125	}	157	Ø	189	¢	221	¦	253	²
31	▼	62	>	94	^	126	~	158	×	190	¥	222	▌	254	■
		63	?	95	_	127	⌂	159	ƒ	191	┐	223	▀	255	space

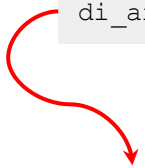
[1]

[1] https://nl.pinterest.com/pin/395261304802336725/

Large Numbers in Data Indexed Arrays

- But still, we can't avoid the problem of treating large numbers.
  - We may need some way to put **arbitrary integers** in a **reasonable size of memory**.

```
di_array.insert(21057381)
```



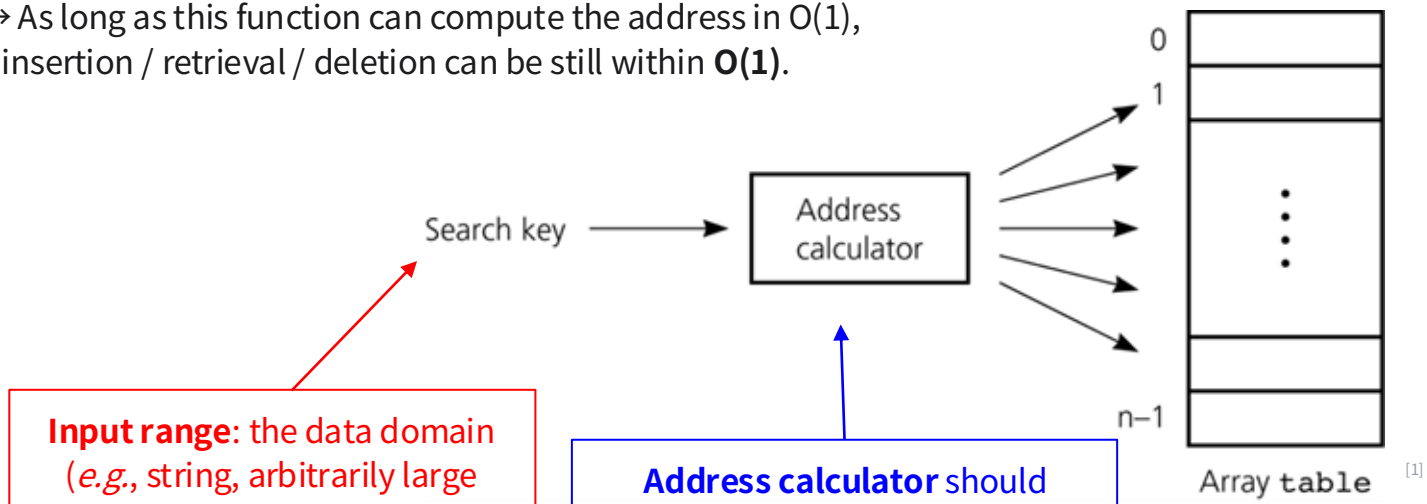
Index	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	…	[999]
Value	F	F	F	F	F	F	F	F	F	⋯	F

Hash Tables

Hash Table

Given a key, a function (called **hash function**) determines where to locate it.

→ As long as this function can compute the address in O(1), insertion / retrieval / deletion can be still within **O(1)**.



[1] https://www.slideshare.net/slideshow/hashing-in-data-structure-is-presented-in-these-slides/274382978

Hash Functions

- A good hash function:
  - Easy and **fast** to compute (in O(1))
  - **Scatter** the data **evenly** on the hash table.
- Selection digits
  - *h*(001364825) = 35
- Folding
  - *h*(001364825) = 001 + 364 + 825 = 1190
- Modulo arithmetic
  - *h*(x) = x `mod` table_size
  - *h*(1004) = 4 (if table_size = 100)
  - This is widely used, as it maps the integers **uniformly** over the internal array regardless of its size!

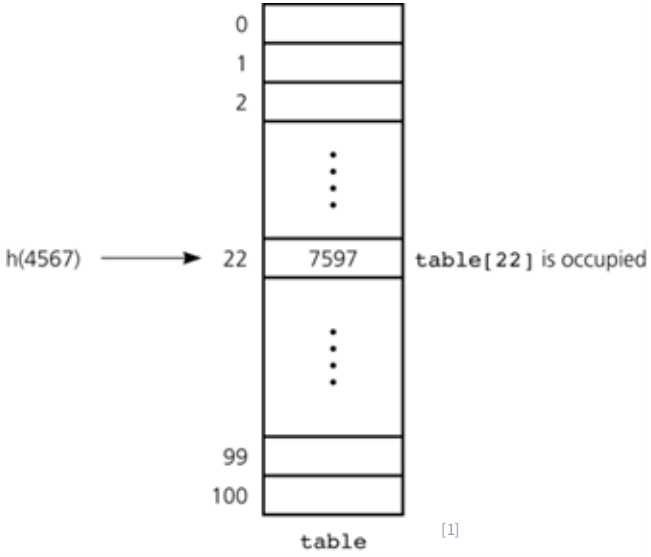
Hmm, then what should we do if two different keys coincidently map to the same address?

Modulo operator (%)
x % y is the remainder when we divide x by y.

upstage Education

Collision

- Yes, unfortunately, it can happen with all hash functions we listed 😨.
  - **Collision**: the phenomenon that two keys are mapped into the same location in the hash table.
- Selection digits
  - $h(001364825) = h(741325385) = 35$
- Folding
  - $h(001364825) = h(825364001) = 1190$
- Modulo arithmetic
  - $h(2205) = h(2405) = 5$ if table_size = 100



[1]

[1] https://www.cs.colostate.edu/~cs165/Spring20/slides/16-Hashing.pdf

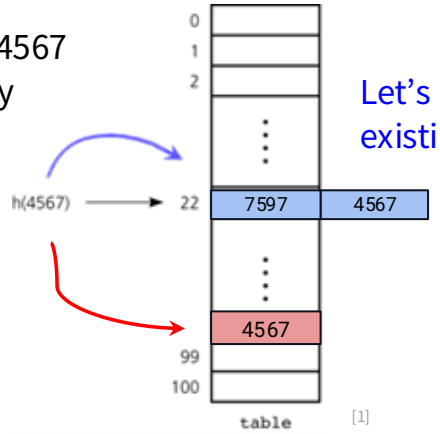
Collision Resolution

- Collision is unavoidable; so, we need a method to resolve it efficiently.
- There are two groups of approaches:
  - **Open addressing**: allowing the items to be located in another place if the collision happened.
  - **Separate chaining**: allowing more than one item to be located at a place.

Example: According to our hash function, 4567 needs to be located at [22], but it is already occupied 😨.

Open Addressing

Let's find another place to put it.



[1]

Separate chaining

Let's put it there together with the existing one(s).

[1] https://www.cs.colostate.edu/~cs165/Spring20/slides/16-Hashing.pdf

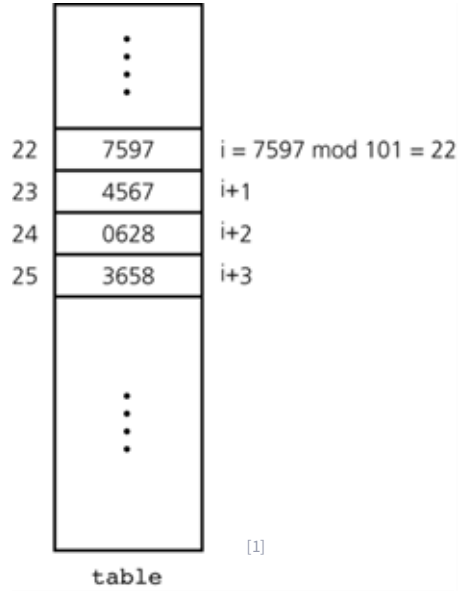
upstage Education

Open Addressing

- **Linear probing**: if the designated spot is occupied, try the right next index, until it finds an available one.

  $h_i(x) = (h_0(x) + i) \bmod \text{table\_size}$

- Causes **primary clustering**:
  - All keys mapped to [22], [23], [24], [25], ⋯ will suffer from collision.
  - If this cluster gets bigger, insertion / retrieval / deletion of keys corresponding to this region will take O(*N*) instead of O(1).
  - Gets significantly worse especially if consecutive numbers are sequentially added. (*e.g.*, 22 → 23 → 24, ⋯)



[1] https://www.cs.colostate.edu/~cs165/Spring20/slides/16-Hashing.pdf

upstage Education

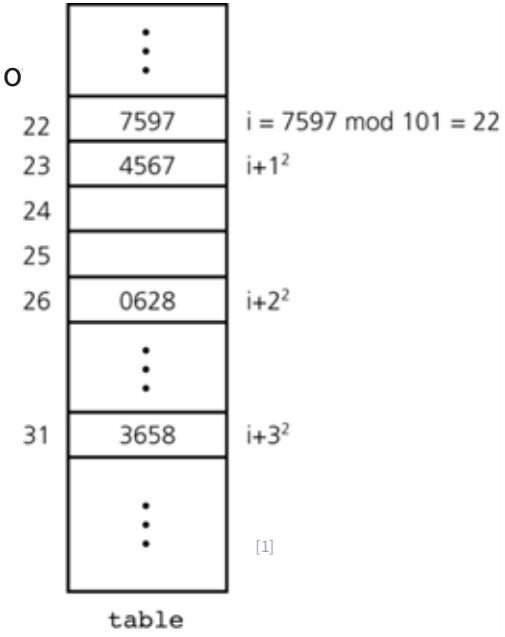
Open Addressing

- **Quadratic probing**: if the designated spot is occupied, try the next set o indices, quadratically increasing with the number of trials.

  $h_i(x) = (h_0(x) + i^2) \bmod \text{table\_size}$

- Solves **primary clustering**:
  - Items collided at [23] suffers less from significant collisions originally happened at [22].

- Still suffers from **secondary clustering**:
  - For items collided at the same index for the first time, they must always follow the same trace.



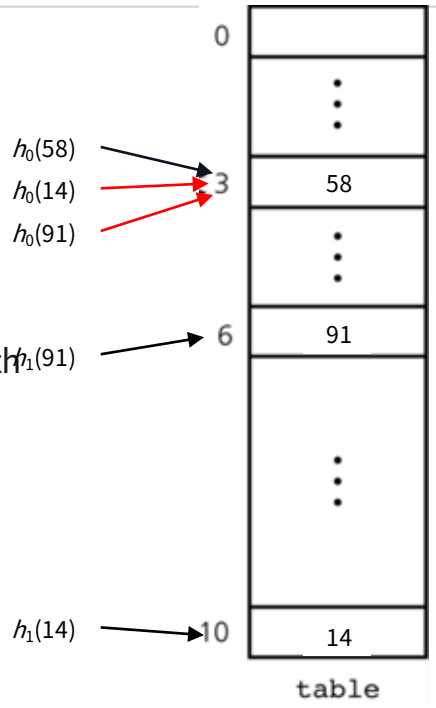
[1] https://www.cs.colostate.edu/~cs165/Spring20/slides/16-Hashing.pdf

Open Addressing

- **Double Hashing**: using two different hash functions to reduce collision probability.

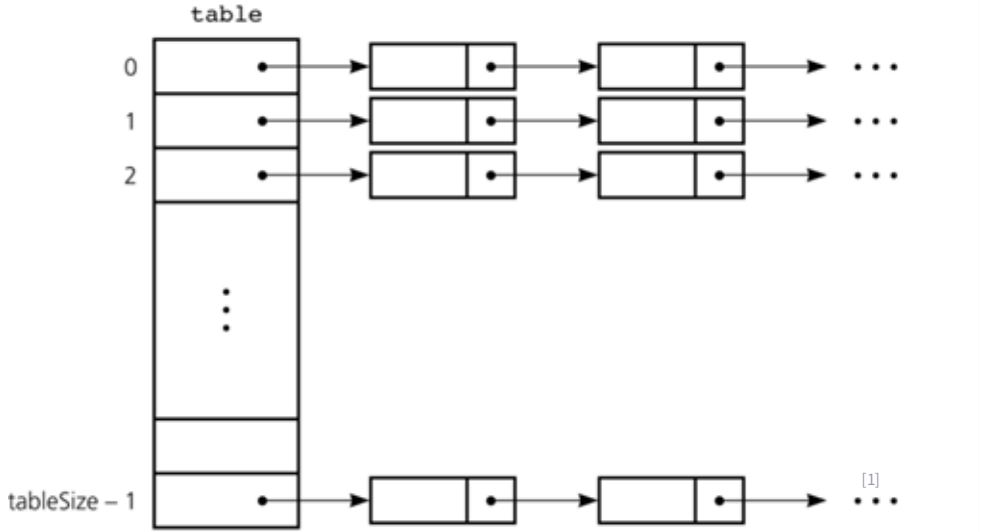
  $h_i(x) = (\alpha(x) + i * \beta(x)) \bmod$ table_size
  where $\alpha(x)$, $\beta(x)$: some hash functions

- Can avoid primary and secondary clustering, since we probe with different steps depending on the input.



Separate Chaining

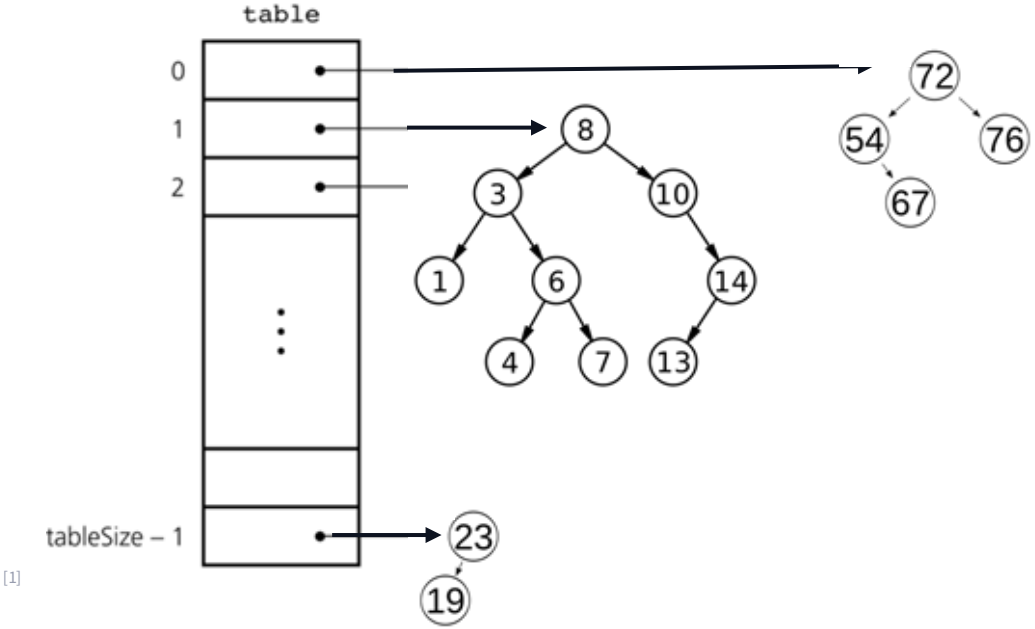
- Each location of the hash table is allowed to have more than one item.
  - A simplest example: a reference to a linked list.



[1] https://medium.com/@ramyjzh/data-structures-for-dummies-hash-tables-579ddd1a4389

Separate Chaining

- Actually, we can put any other data structure that support key-based search.



[1]

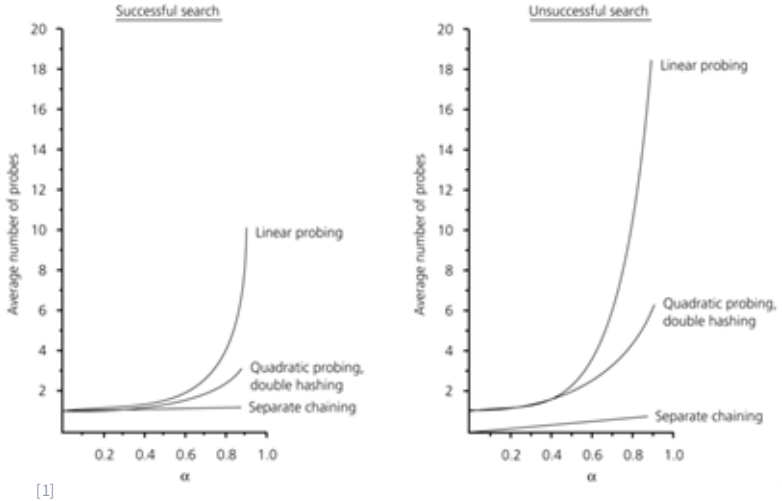
[1] https://medium.com/@ramyjzh/data-structures-for-dummies-hash-tables-579ddd1a4389

Performance of a Hash Table

- In general, having more items in the table reduces performance of a hash table.
  - This is unique to hash tables, unlike most other data structures!
- Let's define **load factor (α)** formally: N/M
  - N: the actual number of elements inserted
  - M: the number of indices (table_size)
- With **open addressing**, α is always between 0 and 1. Once it reaches to 1, we can't add more data because there is no more available space.
  - If α gets **close to 1**, there are probably lots of collisions, meaning **longer processing time**.
- With **separate chaining**, load factor can be > 1.
  - In this case, load factor means the average chain length per index, which is equivalent to the expected number of linear traversal to locate an item.
  - Obviously, higher load factor means slower data processing.

upstage Education

Efficiency of Hashing



[1]

When the load factor is low, hash table achieves ≈O(1), regardless of the collision resolution method.

[1] https://www.eecs.yorku.ca/course_archive/2003-04/F/2011/2011A/DatStr_152_HashTables.pdf

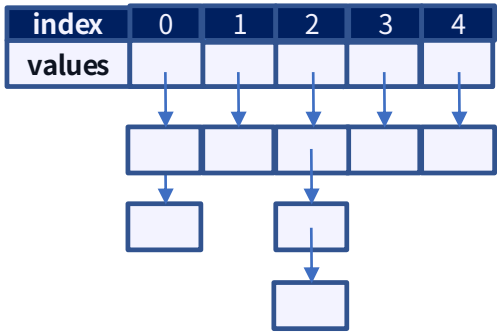
How to Decide the Table Size (M)

- If we know the data size (N) to be inserted, we can set $M = N/\alpha$, where α is the desired load factor.
  - With small α, we focus more on speed.
  - With large α, we care more about the memory space.
  - It is known that α **= 0.6~0.75** is a good balance between them.

- What if we don't know the data size?
  - We have no choice but to dynamically adapt.
  - Starting with some default size (say, 101), then (roughly) double it when the load factor becomes larger than some threshold (*e.g.*, 0.6~0.75).

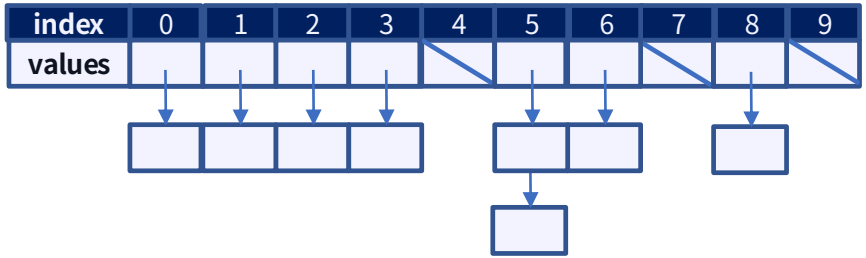
upstage Education

Resizing Example



N=0	N=1	N=2	N=3	N=4
M=5	M=5	M=5	M=5	M=5
α=0	α=0.2	α=0.4	α=0.6	α=0.8

N=5	N=6	N=7	N=8
M=5	M=5	M=5	M=5
α=1.0	α=1.2	α=1.4	α=1.6



Resizing!

Redistributing all items!

N=8
M=10
α=0.8

Time Complexity of Resizing

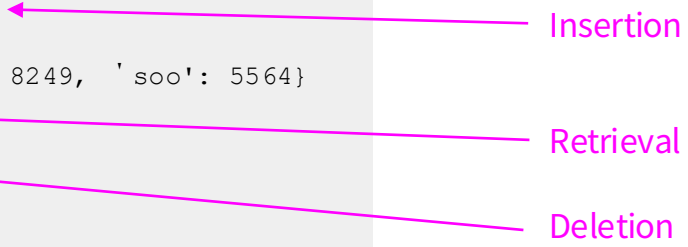
- Besides resizing, hash table achieves O(1) insertion, deletion, and retrieval.
- Resizing is obviously not free 😅
  - Resizing a hash table with N items requires **$O(N)$** time to redistribute all N items.
- A good news 😀: we don't always resize since one resizing operation doubles the number of indices.
  - The number of redistributing items while inserting N items:
    - $1 + 2 + 4 + 8 + \cdots + N = 2N - 1$
  - Overall, redistributing cost becomes $O((2N-1)/N)$ = **O(1) on average**!

Implementations: Dictionary vs Set in Python

Python Dictionary

- Python internally provides the hash table under the name of `Dictionary`.
  - **Key-value pairs** are stored in a hash table.
  - Key are unique and hashable.
  - Insertion, retrieval, deletion supported in **O(1)**.

```
>>> tel = {'jerry': 1086, 'jose': 8249}
>>> tel['soo'] = 4127
>>> tel
{'jerry': 1086, 'jose': 8249, 'soo': 5564}
>>> tel['jose']
8249
>>> del tel['jerry']
>>> tel['shawn'] = 8080
>>> tel
{'jose': 8249, 'soo': 5564, 'shawn': 8080}
>>> list(tel)
['jose', 'soo', 'shawn']
>>> sorted(tel)
['shawn', 'soo', 'jose']
>>> 'jinri' in tel
False
>>> 'jeongwoo' not in tel
True
```



Python Set

- If you don't have an associated `value`, you may use Python `Set`, instead of `Dictionary`.
  - **Only elements (keys)** are stored in the hash table.
  - Elements are unique and hashable.
  - Insertion, retrieval, deletion supported in **O(1)**.

```
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a
{'a', 'r', 'b', 'c', 'd'}    Unique letters in a
>>> a - b
{'r', 'd', 'b'}  Set difference
>>> a | b
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}    Set OR
>>> a & b
{'a', 'c'}  Set AND
>>> a ^ b
{'r', 'd', 'b', 'm', 'z', 'l'}    Set XOR
```

Applications of Hash Tables

Problem 1 - Palindrome

409. Longest Palindrome

Easy | Topics | Companies

Given a string `s` which consists of lowercase or uppercase letters, return the length of the **longest palindrome** that can be built with those letters.

Letters are **case sensitive**, for example, `"Aa"` is not considered a palindrome.

Example 1:

```
Input: s = "abccccdd"
Output: 7
Explanation: One longest palindrome that can be built is "dccaccd", whose length is 7.
```

Example 2:

```
Input: s = "a"
Output: 1
Explanation: The longest palindrome that can be built is "a", whose length is 1.
```

Problem 2 - Smallest Missing Integer

- Given a list of integers, return the smallest positive integer which is not in that list.

Examples:

- [7, 2, 3, 5, 4, 1] → 6
- [-1, 5, 2, 3, 9] → 1
- [17, 25, 4308, 1, 99] → 2

```
# TODO(students): implement this!
def smallest_missing_pos_int(self, list):
  for each item in the list:
    insert into a hash table (hastset)

  for i = 1, 2, 3, ...:
    if the hash table contains i: keep going
    else: return i
```

Note: think about how many times we should iterate here.

upstage Education

Problem 3 – Two Sum

1. Two Sum

Solved

Easy | Topics | Companies | Hint

Given an array of integers `nums` and an integer `target`, return *indices of the two numbers such that they add up to* `target`.

You may assume that each input would have ***exactly* one solution**, and you may not use the *same* element twice.

You can return the answer in any order.

Example 1:

```
Input: nums = [2,7,11,15], target = 9
Output: [0,1]
Explanation: Because nums[0] + nums[1] == 9, we return [0, 1].
```

Example 2:

```
Input: nums = [3,2,4], target = 6
Output: [1,2]
```

Example 3:

```
Input: nums = [3,3], target = 6
Output: [0,1]
```

upstage

Building intelligence for the future of work