Review

- Compared to tree, graph allows **cycles** and **disconnection**
- Graph traversals are a bit different from tree traversals but their core ideas are the same
  - Tracking what nodes have been visited to overcome cycles
  - Outer for loop to cover all disconnected islands
- Graph traversals are the most common tools for solving graph problems
- Please draw memory model for clearer understanding!! ☺

Hash

Lecture 19

Hyung-Sin Kim

SNU Graduate School of Data Science

Contents

- Data Indexed Array
  - Data Indexed Array
  - Data Indexed Array with Chains
- Hash
  - Hash Table
  - Hash Table – Resizing
  - Hash function

Data Indexed Array

- **Data Indexed Array**
- **Data Indexed Array with Chains**

A Magic in Sets and Dictionaries

- As we’ve seen, searching an element in a list takes
  - O(N) for linear search
  - O(logN) for binary search, which requires the list to be **sorted**

- But somehow, searching an element in a set or a dictionary takes
  - O(1)

- We know that this is due to **hashing** which we don’t know yet

Let's dive into its implementation
to see how O(1) is possible!

Data Indexed Arrays

- A regular list: [2, 5, 9, 10]

index	0	1	2	3
values	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	11
values	F	F	T	F	F	T	F	F	F	T	T	F

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
  - Let's assume that Python **Sets** are implemented based on **data-indexed array**
  - di_array = set()   # assuming that the array can have only 0 to 11 as its data

index	0	1	2	3	4	5	6	7	8	9	10	11
values	F	F	F	F	F	F	F	F	F	F	F	F

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
  - Let's assume that Python **Sets** are implemented based on **data-indexed array**
  - di_array = set()   # assuming that the array can have only 0 to 11 as its data
- When data **x** is added to the array, x-th element (di_array[x]) becomes **True**
  - di_array.add(2)
  - di_array.add(9)

index	0	1	2	3	4	5	6	7	8	9	10	11
values	F	F	T	F	F	F	F	F	F	T	F	F

Data Indexed Arrays

- Now, **in** operation (x **in** di_array) simply checks if x-th element is **True**
  - It can just return the value of **di_array[x]**: **O(1)!**

index	0	1	2	3	4	5	6	7	8	9	10	11
values	F	F	T	F	F	F	F	F	F	T	F	F

Ok… but indices are all integers…

Can we also store **string data** in data-indexed arrays?

English (lower case) in Data-Indexed Arrays

- We want to add English words to a data-indexed array
  - di_array.add("gsds")
  - di_array.add("snu")
- What and where are "gsds"-th and "snu"-th elements?
  - Map each of 26 English alphabets to an integer (a=1, b=2, …, z=26)
    - "gsds" becomes $\mathbf{gsds_{27}} = (7 \times 27^3) + (19 \times 27^2) + (4 \times 27^1) + (19 \times 27^0) = 151{,}759$
    - "snu" becomes $\mathbf{snu_{27}} = (19 \times 27^2) + (14 \times 27^1) + (19 \times 27^0) = 14{,}248$
  - Every lower-case word can be represented as a **unique integer**!

"gsds"? "snu"?

index	0	1	2	3	4	5	6	7	8	9	10	11
values	F	F	F	F	F	F	F	F	F	F	F	F

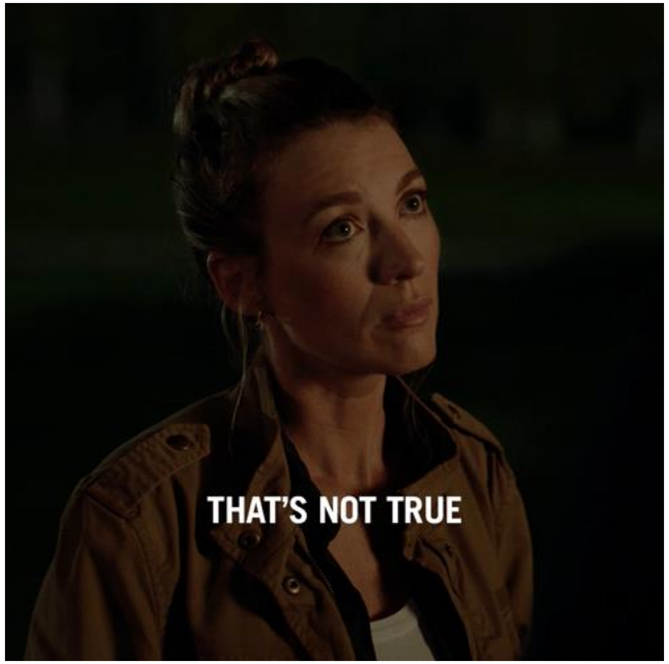
String in Data-Indexed Arrays

- Recall that each character can be represented as an ASCII code value (1~255)
- “8a!”
  - 8: 56
  - a: 97
  - !: 33
- $\mathbf{8a!}_{\mathbf{256}} = (56 \times 256^2) + (97 \times 256^1) + (33 \times 256^0)$
- A string can be represented as a **unique integer** again!

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	è	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	Ô
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	È	176	░	208	ð	240	-
18	↕	49	1	81	Q	113	q	145	æ	177	▒	209	Đ	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	t	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	Ì	254	■
		63	?	95	_	127	⌂	159	ƒ	191	┐	223	▀	255	space

Data-indexed arrays we've seen so far
are great and represent all strings,

if a computer can represent
infinite number of integers and have infinite memory



Integer Overflow

- Python 3 does not have a maximum integer but C/C++/Java does have maximum (unsigned) integer: 4,294,967,295 (2^{32}-1)
  - That means an integer **n** larger than the maximum value **M** is represented as **n % M**, instead of n itself

- $\mathbf{snu\_gsds}_{256}$
  - $115 \times 256^7 + 110 \times 256^6 + 117 \times 256^5 + 95 \times 256^4 + 103 \times 256^3 + 115 \times 256^2 + 100 \times 256^1 + 115 \times 256^0$
  - 8,317,714,614,417,843,315 >> 4,294,967,295

- A data index can easily be larger than the maximum integer
- We should represent an information as one of 4,294,967,296 integers!

Hash Function

- Wikipedia
  - “A hash function is any function that can be used to map **data of arbitrary size** to **fixed-size values**.”

- When we have more than 4,294,967,296 data, **collisions** are inevitable!
  - Ex.) Our hash values for “ace!” and “pace!” are both 1,633,903,905

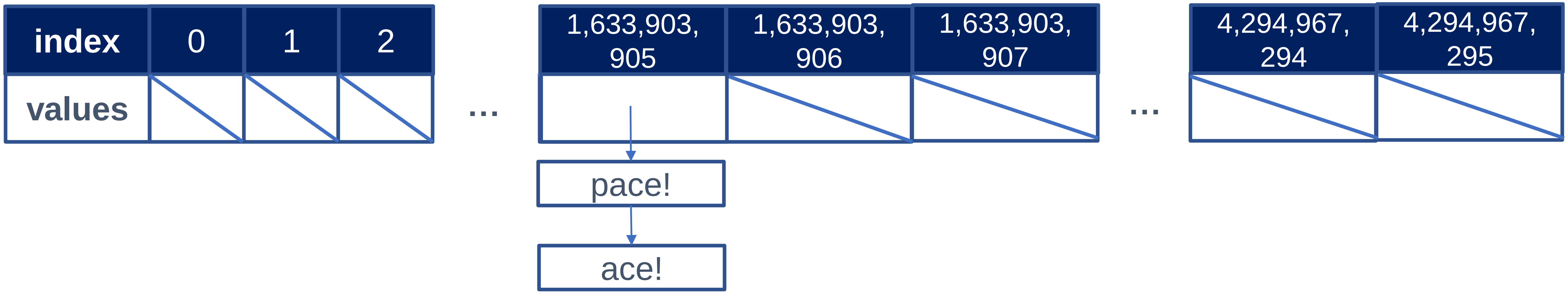
- Two questions
  - How can we handle when hash values are collided?
  - How can we compute a hash function?

Collision Handling

- di_array[x] should contain a list of data whose hash value is **x**
  - Ex.) di_array[1,633,903,905] should contain "pace!" and "ace"

index	0	1	2	…	1,633,903, 905	1,633,903, 906	1,633,903, 907	…	4,294,967, 294	4,294,967, 295
values	F	F	F	…	F	F	F	…	F	F

- To this end, we can make di_array[x] as a **linked list** instead of one value
  - Using an array or a set is also possible



Data Indexed Array

- Data Indexed Array
- **Data Indexed Array with Chains**

Data-Indexed Array with Chains

- Each element is initially **None** but becomes a linked list when an item is added

```
def __init__(self) -> None:
    self.array = [None]*4294967296

def add(self, x) -> None:
    i = hash_value(x)
    if self.array[i] == None:
        self.array[i] =  SLList()
    self.array[i].addFirst(x)
```

Data-Indexed Array with Chains

- Each element is initially **None** but becomes a linked list when an item is added

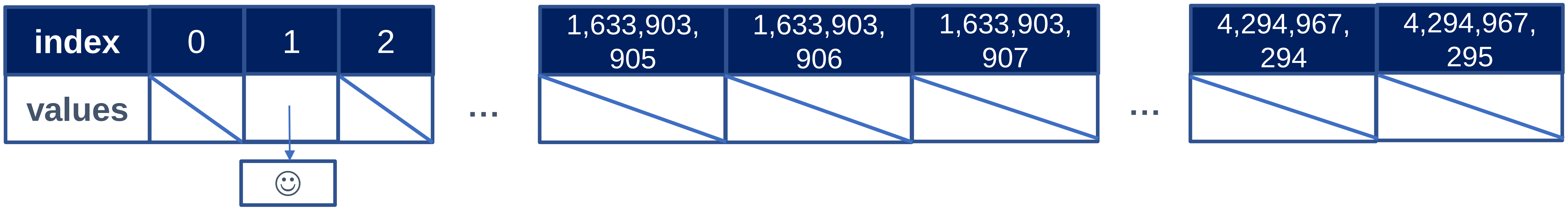
```
def __init__(self) -> None:
    self.array = [None]*4294967296

def add(self, x) -> None:
    i = hash_value(x)
    if self.array[i] == None:
        self.array[i] =  SLList()
    self.array[i].addFirst(x)
```

```
A = di_array()
```

index	0	1	2	…	1,633,903,905	1,633,903,906	1,633,903,907	…	4,294,967,294	4,294,967,295
values										

Data-Indexed Array with Chains

- Each element is initially **None** but becomes a linked list when an item is added
  - def **__init__**(self) -> None:
  -     self.array = [None]*4294967296
  - def **add**(self, x) -> None:
  -     i = hash_value(x)
  -     if self.array[i] == None:
  -         self.array[i] = SLList()
  -     self.array[i].addFirst(x)
  - A = di_array()
  - A.add(“☺”)

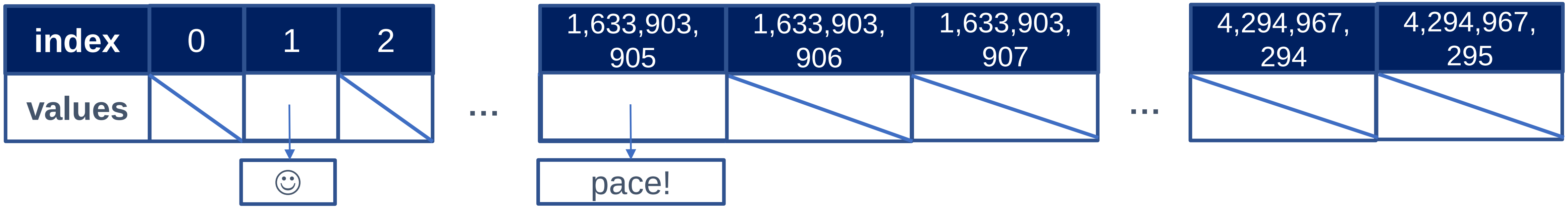


Data-Indexed Array with Chains

- Each element is initially **None** but becomes a linked list when an item is added

```
def __init__(self) -> None:
    self.array = [None]*4294967296

def add(self, x) -> None:
    i = hash_value(x)
    if self.array[i] == None:
        self.array[i] =  SLList()
    self.array[i].addFirst(x)
```

```
A = di_array()
A.add("☺")
A.add("pace!")
```

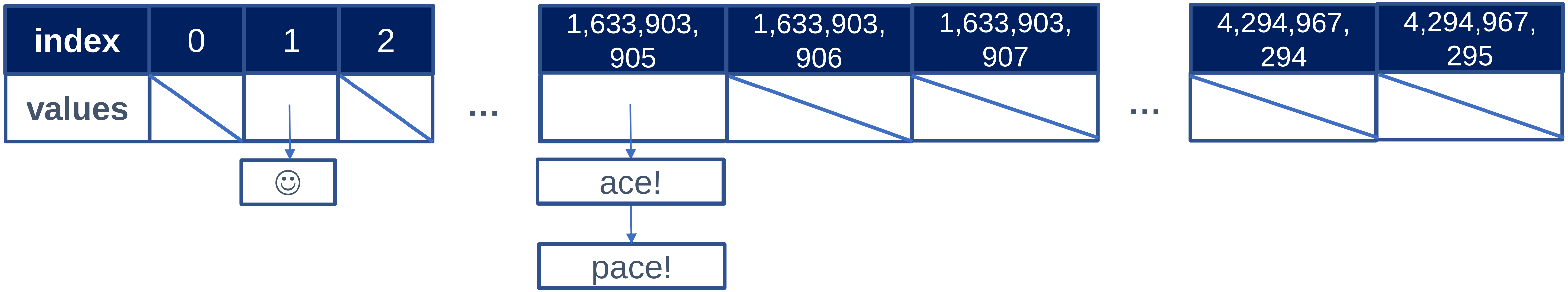


Data-Indexed Array with Chains

- Each element is initially **None** but becomes a linked list when an item is added

```
def __init__(self) -> None:
    self.array = [None]*4294967296

def add(self, x) -> None:
    i = hash_value(x)
    if self.array[i] == None:
        self.array[i] =  SLList()
    self.array[i].addFirst(x)
```

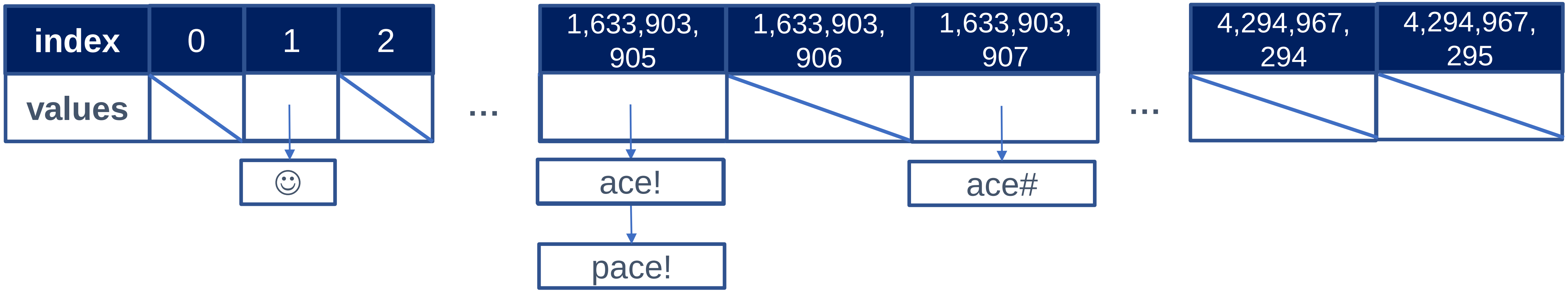
```
A = di_array()
A.add("☺")
A.add("pace!")
A.add("ace!")
```



Data-Indexed Array with Chains

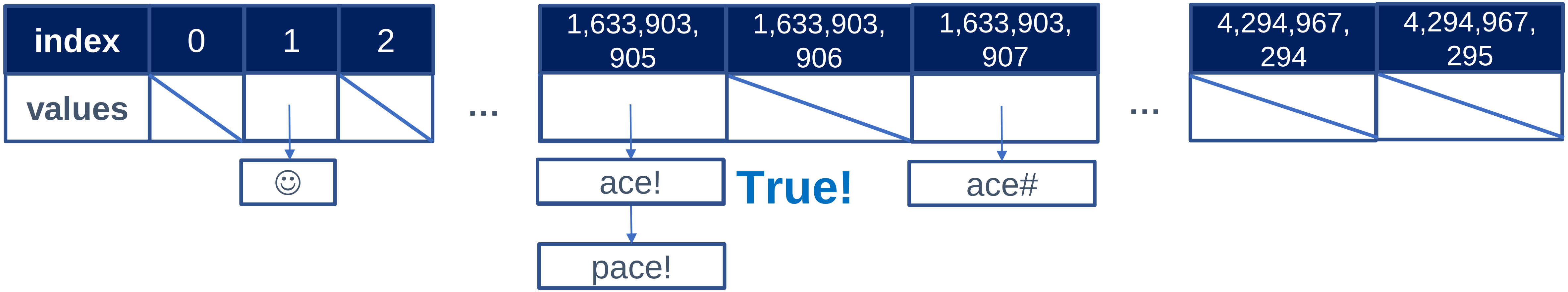
- Each element is initially **None** but becomes a linked list when an item is added
  - def **__init__**(self) -> None:
  -     self.array = [None]*4294967296
  - def **add**(self, x) -> None:
  -     i = hash_value(x)
  -     if self.array[i] == None:
  -         self.array[i] = SLList()
  -     self.array[i].addFirst(x)

  - A = di_array()
  - A.add(“☺”)
  - A.add(“pace!”)
  - A.add(“ace!”)
  - A.add(“ace#”)



Data-Indexed Array with Chains

- Each element is initially **None** but becomes a linked list when an item is added
  - def **__init__**(self) -> None:
  -     self.array = [None]*4294967296
  - def **add**(self, x) -> None:
  -     i = hash_value(x)
  -     if self.array[i] == None:
  -         self.array[i] = SLList()
  -     self.array[i].addFirst(x)
  - A = di_array()
  - A.add(“☺”)
  - A.add(“pace!”)
  - A.add(“ace!”)
  - A.add(“ace#”)
  - “ace!” in A
    - Check all items in A.array[1633903905]



Hash

- **Hash Table**
- **Hash Table – Resizing**
- **Hash Function**

Finally… Hash Table

- Now that we can handle collisions, we don't actually need all 4,294,967,296 indices

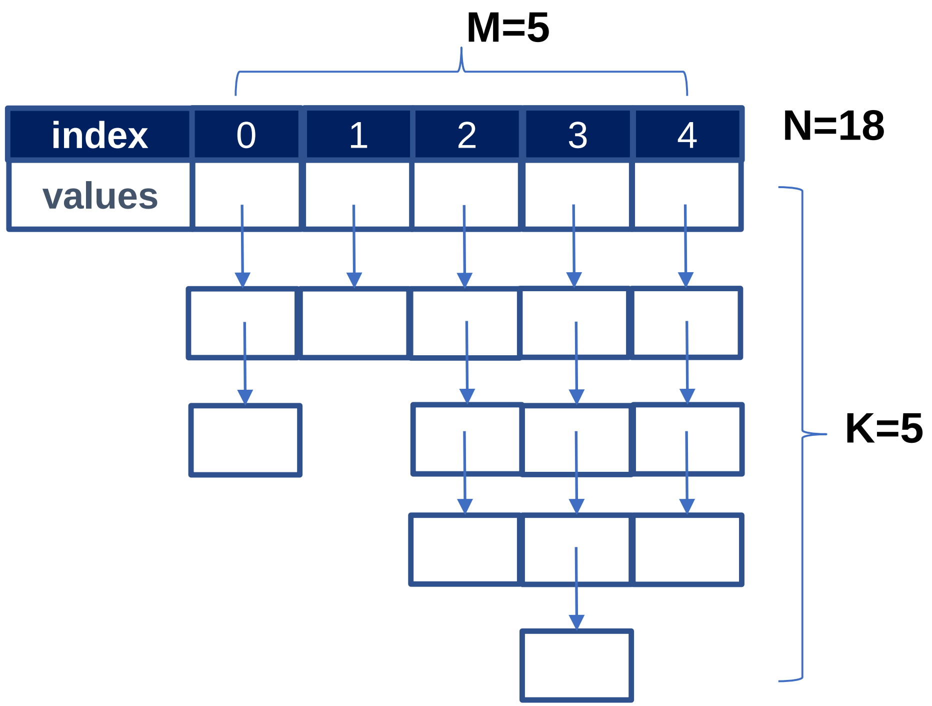
- What if we have only 12 indices?
  - When adding item **x**, compute its hashValue **i**
  - Add x to di_array[**i%12**] instead of di_array[i]

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

- **Hash table**
  - A table that stores data by using a valid index that is computed as follows:
  - Data ⟹ hash function ⟹ hash value ⟹ reduction (e.g., modulo) ⟹ valid index

Hash Table Performance

- With a few indices, now we don't waste memory
- On the flip side, time cost is proportional to length of the longest chain
  - K: length of the longest chain
  - What is **K**? Given M and N, how can we reduce **K**?

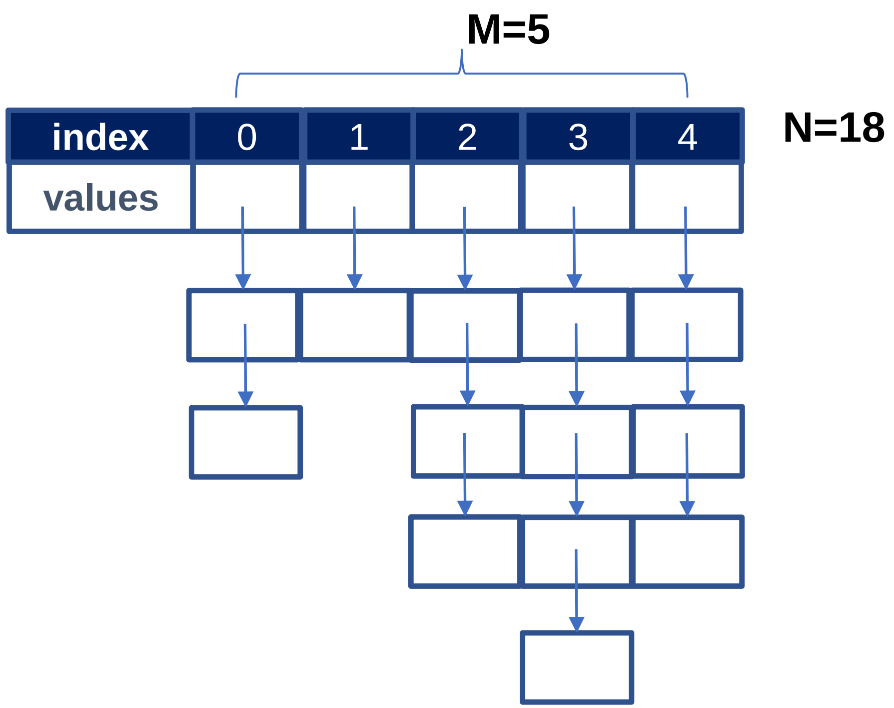


Case	Add	in
SLList	O(1)	O(N)
List	O(1)	O(N)
Data-indexed Array	O(1)	O(1)
Data-indexed Array with Chains	O(1)	O(K)

Hash Table Performance – Problems

- Assume that our hash table has M indices and N items
  - K is between N/M (best case, evenly spread) and N (worst case, a long single chain)

- But the **real problem** is…
  - When M is **fixed** (e.g., 5), O(K) = O(N/5) = O(N)
  - Even in the base case, time cost increases with **N**!



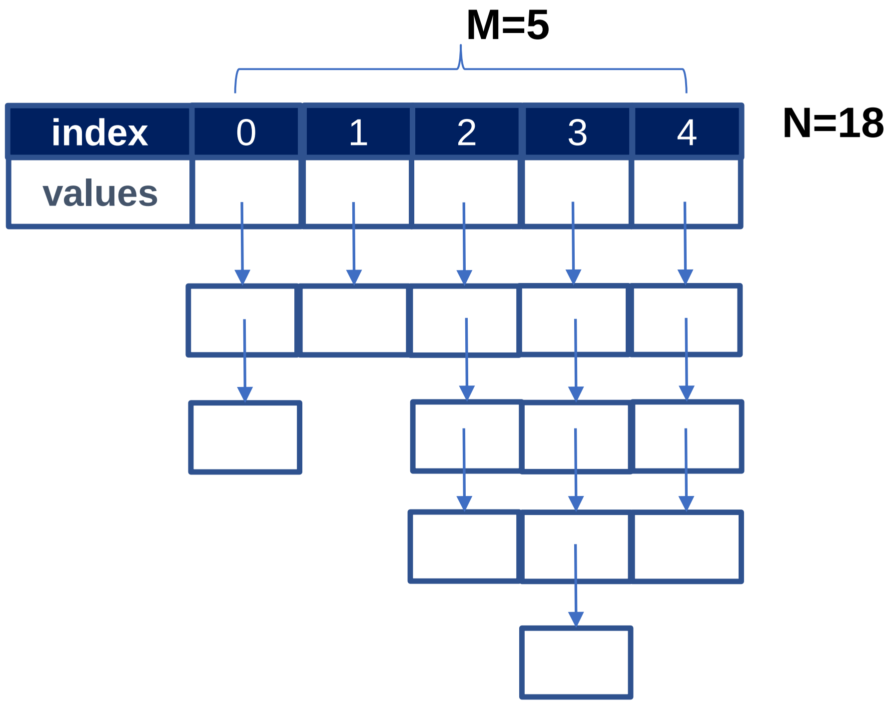
How can we improve our hash table to achieve **O(1)**?

Hash

- Hash Table
- **Hash Table – Resizing**
- Hash Function

Hash Table Performance – Resizing

- Instead of using a fixed M, we can increase **M** as **N** increases
  - If we increase M proportional to N, **O(N/M)** becomes **O(1)**!
- For example, we can double **M** when **N/M >= 1.5**
- Then, the hash table's chains now have **less than 1.5 items** on average



Hash Table Performance – Resizing

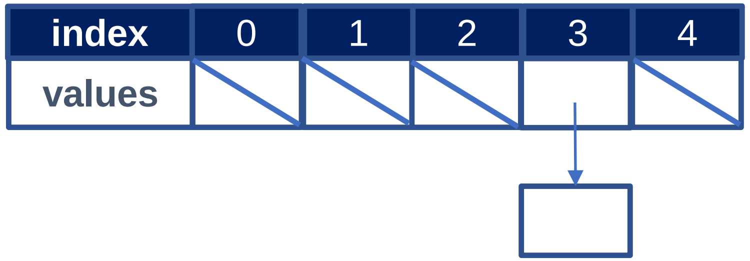
index	0	1	2	3	4
values					

N=0

M=5

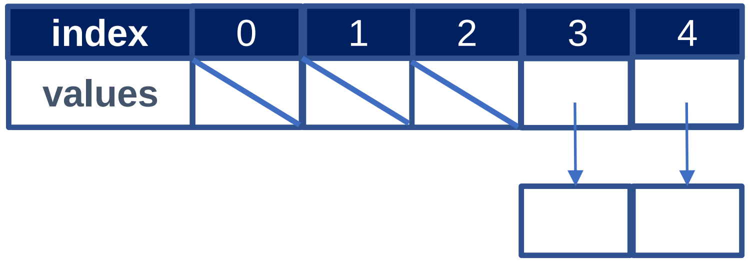
N/M=0

Hash Table Performance – Resizing



N=0	**N=1**
M=5	**M=5**
N/M=0	**N/M=0.2**

Hash Table Performance – Resizing

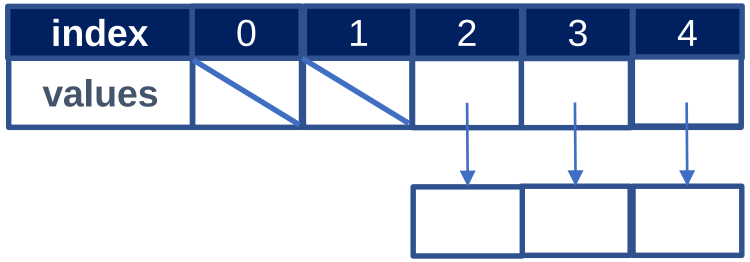


N=0 M=5 N/M=0

N=1 M=5 N/M=0.2

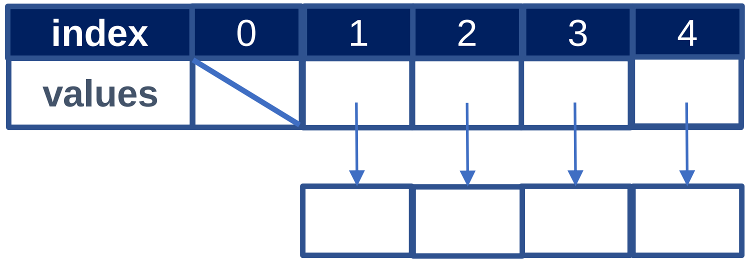
N=2 M=5 N/M=0.4

Hash Table Performance – Resizing



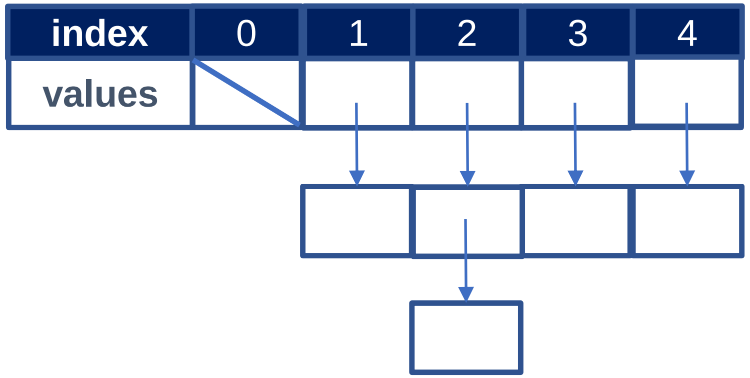
N=0	N=1	N=2	N=3
M=5	M=5	M=5	M=5
N/M=0	N/M=0.2	N/M=0.4	N/M=0.6

Hash Table Performance – Resizing



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

Hash Table Performance – Resizing



N=0
M=5
N/M=0

N=1
M=5
N/M=0.2

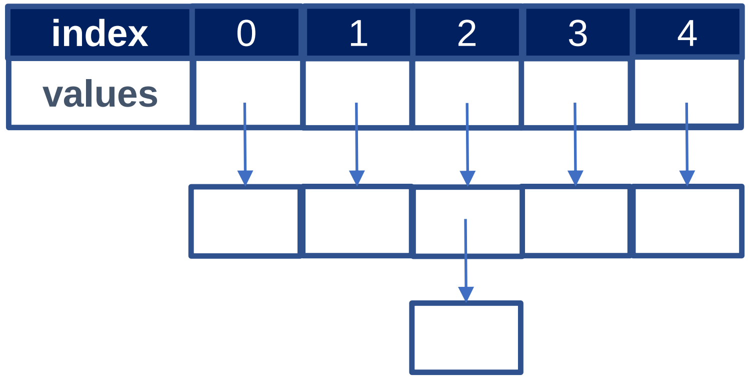
N=2
M=5
N/M=0.4

N=3
M=5
N/M=0.6

N=4
M=5
N/M=0.8

N=5
M=5
N/M=1.0

Hash Table Performance – Resizing



N=0 M=5 N/M=0

N=1 M=5 N/M=0.2

N=2 M=5 N/M=0.4

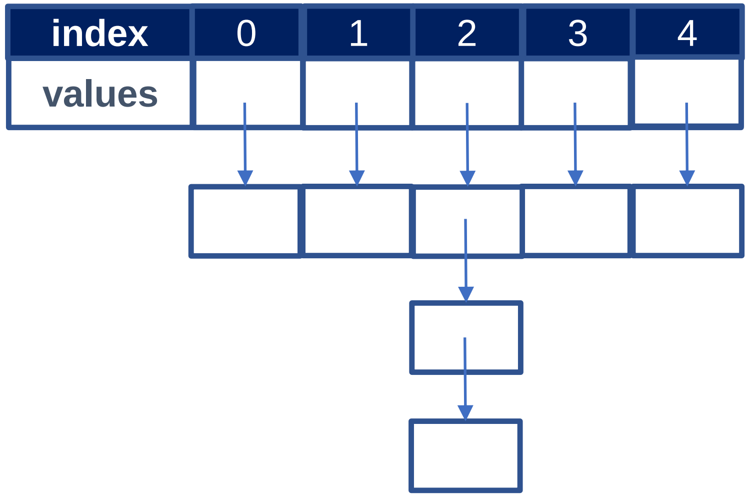
N=3 M=5 N/M=0.6

N=4 M=5 N/M=0.8

N=5 M=5 N/M=1.0

N=6 M=5 N/M=1.2

Hash Table Performance – Resizing



N=0 M=5 N/M=0

N=1 M=5 N/M=0.2

N=2 M=5 N/M=0.4

N=3 M=5 N/M=0.6

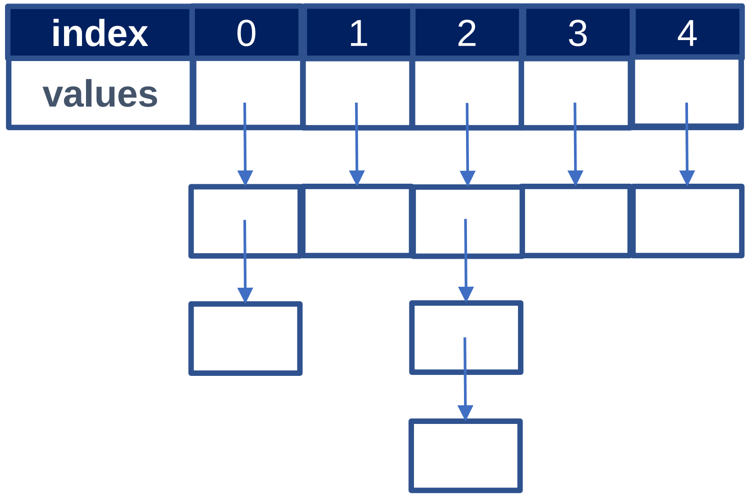
N=4 M=5 N/M=0.8

N=5 M=5 N/M=1.0

N=6 M=5 N/M=1.2

N=7 M=5 N/M=1.4

Hash Table Performance – Resizing



N=0 M=5 N/M=0

N=1 M=5 N/M=0.2

N=2 M=5 N/M=0.4

N=3 M=5 N/M=0.6

N=4 M=5 N/M=0.8

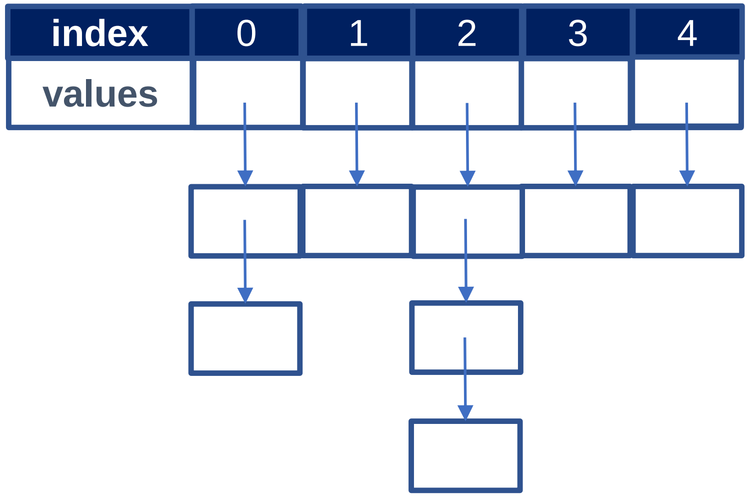
N=5 M=5 N/M=1.0

N=6 M=5 N/M=1.2

N=7 M=5 N/M=1.4

N=8 M=5 N/M=1.6

Hash Table Performance – Resizing



N=0 M=5 N/M=0

N=1 M=5 N/M=0.2

N=2 M=5 N/M=0.4

N=3 M=5 N/M=0.6

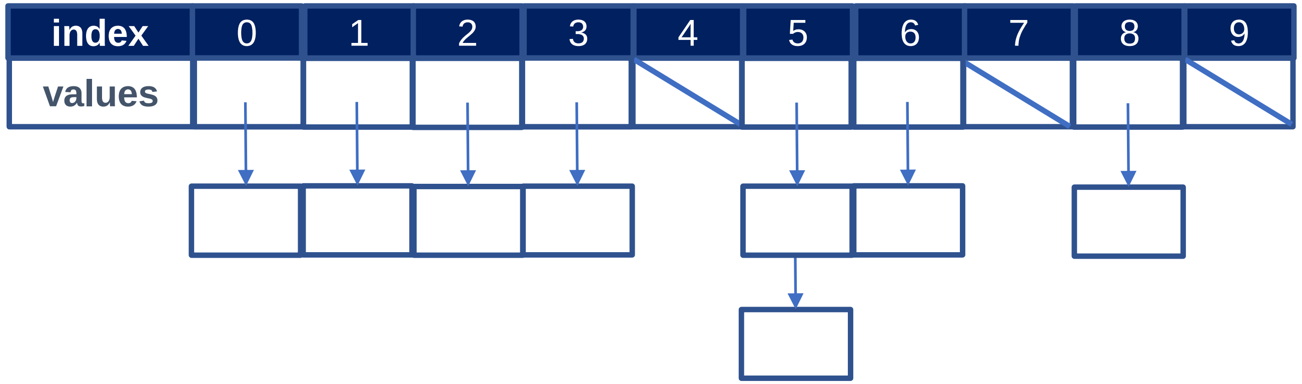
N=4 M=5 N/M=0.8

N=5 M=5 N/M=1.0

N=6 M=5 N/M=1.2

N=7 M=5 N/M=1.4

N=8 M=5 N/M=1.6



Resizing!

Redistributing all items!

N=8

M=10

N/M=0.8

Hash Table Performance – Resizing

- With resizing, searching operation takes only **O(1)**!
  - If resizing operation is free… which is not true

- Resizing a hash table with N items requires **O(N)** time to redistribute all **N** items
  - But a good news is that we **don't always resize** since one resizing operation **doubles** the number of indices

- The number of redistributing items while adding N items
  - 1 + 2 + 4 + 8 + …. + N = **2N-1**
  - When adding one item, redistributing cost becomes **O((2N-1)/N) = <u>O(1)</u>** on average!

Hash Table Performance

- Data-indexed array + chaining + resizing
  - Collisions are properly handled
  - Time complexity is independent from **N**
    - If items are **<u>evenly spread</u>** through the whole array … ☺

Case	Add	in
SLList	O(1)	O(N)
List	O(1)	O(N)
Data-indexed Array	O(1)	O(1)
Data-indexed Array with Chains (no resizing)	O(1)	O(N)
Data-indexed Array with Chains (with resizing)	O(1)	O(1)

Data becomes a hash value after passing through a **hash function.**

How can make a hash function that distributes items evenly?

Hash

- Hash Table
- Hash Table – Resizing
- **Hash Function**

Hash Function

- Bad examples
  - def hashfunction(x: str):
  - return 1 *# same index for all strings*

  - def hashfunction(x: str):
  - return ord(x[0]) *# same index for all strings that have the same first character*

  - def hashfunction(x: str):
  - ans = 0
  - for ch in x:
  - ans += ord(ch) *# same index for all strings that consist of same characters*
  - return ans

Hash Function

- Converting a string into a base B number would be good (as we already did)
  - def hashfunction(x: str):
  - ans = 0
  - for ch in x:
  - ans = ans * **B** + ord(ch)
  - return ans

- What is a good base **B**?

Hash Function – Good Base

- Using 256 as a base seems clear since it can give a **unique number** for each string
- But now that we allow collision anyway (due to the limited maximum integer), we don't have to stick to "**uniqueness**"
- Moreover, base 256 causes all strings that share the last four characters collide with each other since the maximum number is $2^{32} = 256^4$
  - "I love **you.**" / "I hate **you.**" / "He likes **you.**" / "It's **you.**"
- Using a **small <u>prime</u> number** as a base is typical

Enough! More details are out of scope of this course

Enjoy sets and dictionaries with a bit more familiarity ☺

Summary

Summary

- Data Indexed Array
  - A data-indexed array has all possible data as its **indices**
  - **in** operation (x **in** di_array) can just return the value of **di_array[x]**: **O(1)!**
- Data Indexed Array with Chains
  - Each element is initially **None** but becomes a linked list when an item is added
- Data Indexed Array with Chains
  - Each element is initially **None** but becomes a linked list when an item is added
- Hash Table
  - Resizing
  - Hash function

Thanks!