

Lecture 11: **Hash ADT**

01204212 Abstract Data Types and Problem Solving

Department of Computer Engineering
Faculty of Engineering, Kasetsart University
Bangkok, Thailand.



Department of
Computer Engineering
Kasetsart University



Outline


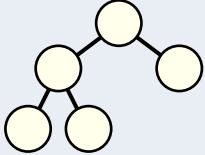
- Hash ADT
- Hash Functions
- Collisions
 - Separate Chaining
 - Open Addressing

The Need for Fast Retrieving Data

- In many applications, we need a system that fast stores and access the data
- For example: a simple system to students' GPA
 - Recall that each student has a unique 10-digit identifier so that we have:
 $(\text{stdID_1}, \text{GPA_1}), (\text{stdID_2}, \text{GPA_2}), \dots, (\text{stdID_n}, \text{GPA_n})$
 - We need the **fast** find(), insert(), and delete() operations

Which ADT?

ADT Candidates

ADT	find()	insert()	delete()
1. List 	$O(n)$	$O(1)$	$O(n)$
2. BST/AVL Tree 	$O(\log n)$	$O(\log n)$	$O(\log n)$

- In real-world applications, n is typically between 100 and 100,000 (or more)
 - $\log n$ is between 6.6 and 16.6
- We need a designed ADT for all operations in $\Theta(1)$ time
 - However, we may incur other costs

Guess! Which data structure?

An Array-Implementation

Our goal:

- Store data so that all operations are $\Theta(1)$ time

Requirement:

- The memory requirement should be $\Theta(n)$
- Normally, the size is **larger** than the number of the data

In practice, we would like to

- Create an array of size M
- Store each of n objects in one of the M bins
- Have some means of **determining the bin in which an object is stored**

Example: Design for Implementation

- Consider 100 records of student ID with his GPA:
(stdID_1, GPA_1), (stdID_2, GPA_2), ..., (stdID_100, GPA_100)
 - We can use the student ID as the index of array
- Suppose “Joe” has the number 6410501234
 - Obviously, we cannot create an array of size 10^{10}
- We could create an array of size 1000:
 - How could you convert a 10-digit number into 3-digit number?
 - First three digits (641...) clearly cause a problem
 - The last three digits, however, are essentially random
- Therefore, Joe’s GPA could be store in `gpa[234]=3.50`

Example: Design for Implementation

Question:

- What is the likelihood that a class of 100 students will have nobody with the same last three digits?

$$\frac{1000}{1000} \cdot \frac{999}{1000} \cdot \frac{998}{1000} \cdot \frac{997}{1000} \cdot \dots \cdot \frac{901}{1000} \approx 0.005959$$

➡ Not very high



Example: Design for Implementation

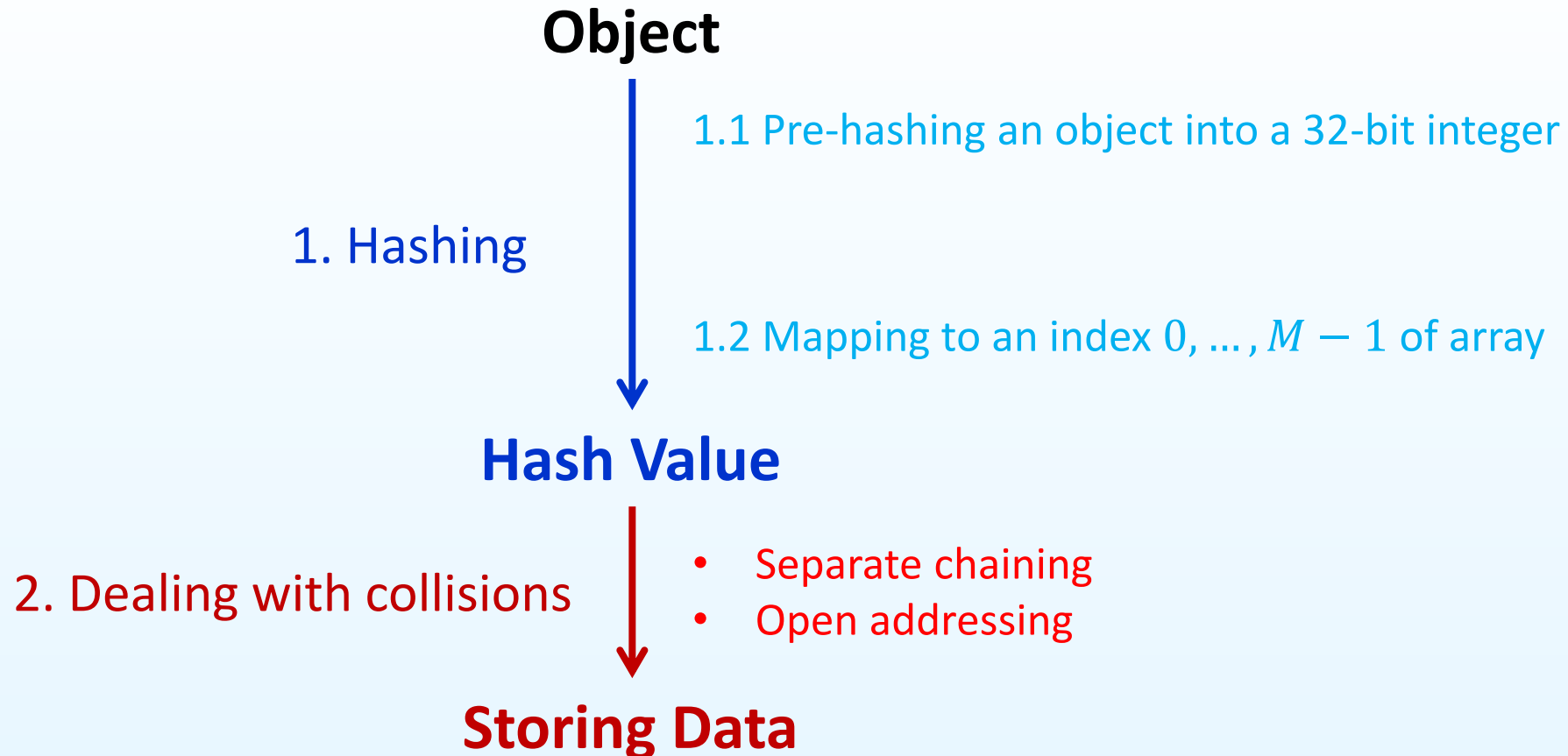
- Consequently, we have a **function** that **maps** a student onto a 3-digit number:
 - The function is the **modulus of 1000**
 - We can store something in that location
 - Storing, accessing, and deleting it are $\Theta(1)$
- However, two or more students may map to the **same** number:
 - Joe has ID 6410501234 and GPA 3.5
 - Alma has ID 6410505234 and GPA 4.0

	⋮
231	
232	
233	
234	3.50
235	
236	
237	
238	
239	3.00
240	
	⋮

Hash Tables

- Objects are stored in an **array**, called a *table*
- The process of **mapping** an object or a number onto an integer in a given range is called *hashing*
- An event that multiple objects may hash to the **same** value is called a *collision*
- Therefore, **hash tables** use a hash function together with a mechanism for dealing with collisions

The Hash Process



Outline

- Hash ADT
- Hash Functions
- Collisions
 - Separate Chaining
 - Open Addressing

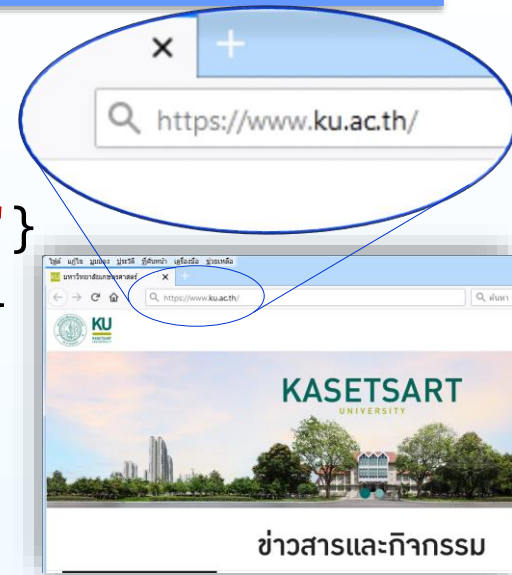
Hashing Schemes

- We want to store n objects in a table of M at a location computed from the key K
- Hash function
 - Method for computing table index from key
- Collision resolution strategy
 - How to handle two keys that hash to the same index

Example: Looking for an IP Address

- Data records:

- {"www.ku.ac.th", "158.108.216.5"}
- {"www.eng.ku.ac.th", "158.108.215.137"}
- {"www.cpe.ku.ac.th", "158.108.32.150"}
- {"adt.mikelab.net", "158.108.32.156"}
- ...

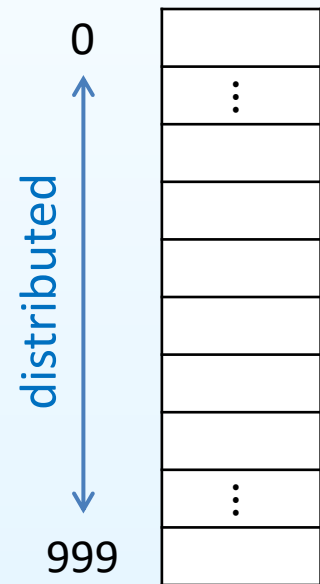


- Can we directly index into the array as the followings?

- table["www.ku.ac.th"] = "158.108.216.5"
- table["www.eng.ku.ac.th"] = "158.108.215.137"
- table["www.cpe.ku.ac.th"] = "158.108.32.150"
- table["adt.mikelab.net"] = "158.108.32.156"
- ...

Mapping into Hash Table

- Need a fast **hash function** h to convert the **element key** (**number** or **string**) to an **integer** (**hash value**)
 - Then, use this value as an index of array
- Given an array `table` of size 1000, we first calculate the index by:
 - $h(\text{"www.ku.ac.th"}) \Rightarrow 851$
 - $h(\text{"www.eng.ku.ac.th"}) \Rightarrow 291$
 - $h(\text{"www.cpe.ku.ac.th"}) \Rightarrow 441$
 - $h(\text{"adt.mikelab.net"}) \Rightarrow 705$
 - ...
- **Output** of the hash function
 - Must always be **less than** size of array
 - Must be as evenly **distributed** as possible



Properties

Necessary properties of such a hash function h are:

- Should be **fast**
 - Ideally $\Theta(1)$
- Must be **deterministic**
 - Always return the same hash value for the same object
- Hash equal objects to equal values
 - $x = y \implies h(x) = h(y)$
- Should be only a one-in-whole chance for hashing two random objects to the same hash value

Steps in Hash Function

A hash function may consist of two steps:

1. Pre-hash an object key into a 32-bit integer
 - The key can be number or string
2. Map the integer to an index of the array
 - Simply modulo the array size

1. Pre-hashing an Object Key

We design a **deterministic hash function** to calculate an **arithmetic hash value** from the relevant member variables of an object

We will look at the examples of arithmetic hash function for

- Rational numbers
 - e.g., (0,1), (1,2), (2,3), (99,100), ...
- Strings
 - e.g., "apple", "boy", "cat", "hello", ...

Example: Pre-hashing Rational Numbers

- Suppose we want to store rational numbers:
 - $(0,1), (1,2), (2,3), (99,100), \dots$
- Obviously, the array may not support tuples as indices
 - We need to first convert an individual into a 32-bit integer
- Each rational number is composed of the numerator (numer) and the denominator (denom)

Example: Pre-hashing Rational Numbers

Version #1

```
unsigned int pre_hash(int numer, int denom) {  
    return (unsigned int)numer + (unsigned int)denom;  
}
```

Problem:

- Rational numbers such as (1,2) and (2,1) have the same value

Example: Pre-hashing Rational Numbers

Version #2

```
#include <stdio.h>
#define MULTIPLIER 429496751 // selected (large) prime
unsigned int pre_hash(int numer, int denom) {
    return (unsigned int)numer +
           MULTIPLIER*(unsigned int)denom;
}
int main(void) {
    printf("%u\n", pre_hash(0,1));
    printf("%u\n", pre_hash(1,2));
    printf("%u\n", pre_hash(2,1));
    printf("%u\n", pre_hash(2,4));
    printf("%u\n", pre_hash(99,100));
    return 0;
}
```

429496751
858993503
429496753
1717987006
2239

Arithmetic operations wrap on overflow

Problem:

- Rational numbers such as (1,2) and (2,4) have different value



Example: Pre-hashing Rational Numbers

Version #3

```
#include <stdio.h>
#define MULTIPLIER 429496751 // selected (large) prime
unsigned int pre_hash(int numer, int denom) {
    int divisor = gcd(numer, denom);
    divisor = (denom >= 0)? divisor : -divisor;
    numer /= divisor;
    denom /= divisor;
    return (unsigned int)numer +
           MULTIPLIER*(unsigned int)denom;
}
int main(void) {
    printf("%u\n", pre_hash( 1, 2));
    printf("%u\n", pre_hash(-1, 2));
    printf("%u\n", pre_hash( 2,-4));
    printf("%u\n", pre_hash(-2,-4));
    return 0;
}
```

Need to be implemented

Make the denominator always positive

```
858993503
858993501
858993501
858993503
```



Example: Pre-hashing Strings

- Suppose we want to store words:
 - “apple”, “boy”, “cat”, “hello”, ...
- Also, in C Program, a string cannot be an index of array
 - We need to first convert an individual into a 32-bit integer
- A string is simply an **array of bytes**
 - Each byte stores a value from 0 to 255
 - A hash function can be a function of these bytes

Example: Pre-hashing Strings

Version #1

```
unsigned int pre_hash(char *str) {  
    unsigned int hash_value = 0;  
    int          i = 0;  
  
    for (i=0; i<strlen(str); i++)  
        hash_value += str[i];  
    return hash_value;  
}
```

Problem:

- Words with the same characters hash to the same location: “from” and “form”
- Slow running time: $\Theta(n)$

Example: Pre-hashing Strings

Version #2.1

```
#define MULTIPLIER 12347 // selected prime
unsigned int pre_hash(char *str) {
    unsigned int hash_value = 0;
    int i = 0;

    for (i=0; i<strlen(str); i++)
        hash_value = MULTIPLIER*hash_value + str[i];
    return hash_value;
}
```

Let the individual characters represent the **coefficients** of a polynomial in x :

$$p(x) = c_0x^{n-1} + c_1x^{n-2} + \dots + c_{n-3}x^2 + c_{n-2}x + c_{n-1}$$

Example: Pre-hashing Strings

Version #2.2

```
#define MULTIPLIER 12347 // selected prime
unsigned int pre_hash(char *str) {
    unsigned int hash_value = 0;
    int i = 0;

    for (i=1; i<=strlen(str); i*=2)
        hash_value = MULTIPLIER*hash_value + str[i-1];
    return hash_value;
}
```

We may pick some characters only:

- Use the characters in locations $2^k - 1$ for $k = 0, 1, 2, \dots$

0, 1, 3, 7, 15, ...

2. Mapping to an Array Index

Suppose the array has size M , the easiest method is to return the value **modulus M**

```
unsigned int hash_map(unsigned int hv, unsigned int M) {  
    return hv % M;  
}
```

Note that the modulus operator % is relative slow, we may use the bitwise operators such as &, <<, or >> instead

Outline

- Hash ADT
- Hash Functions
- Collisions
 - Separate Chaining
 - Open Addressing

Collisions

- A collision occurs when two different object keys hash to the **same value**
 - For example, for a table size 17, the object keys 18 ($18\%17=1$) and 35 ($35\%17=1$) will hash to the same value
- We cannot store the data records in the same slot of the array

Collision Resolutions

1. Separate Chaining

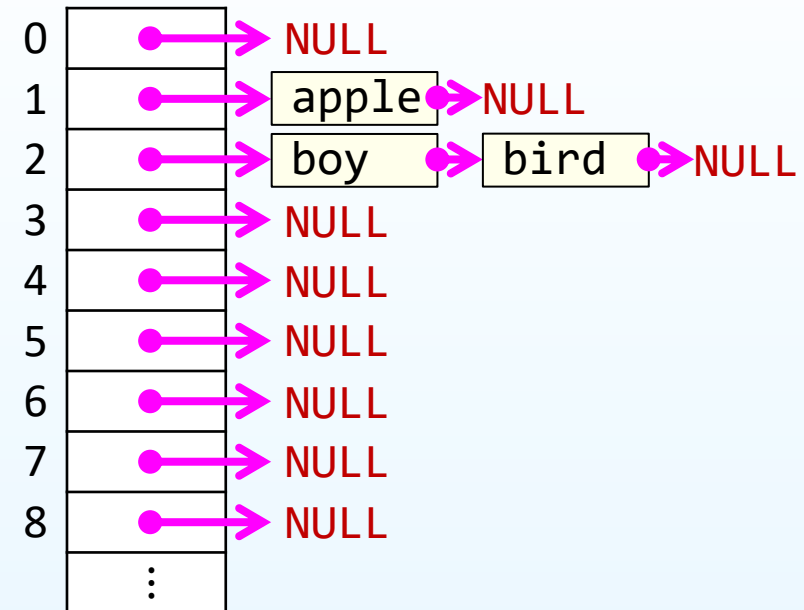
- Use **data structure** (such as a linked list) to store multiple objects that hash to the same slot

2. Open addressing (or probing)

- Search for empty slots using **a second function** and store object in first empty slot that is found

1. Resolution by Separate Chaining

- Each hash table cell holds **pointer** to a linked list
- Collision**: insert object into the linked list of same hash value
- To **find** an object: compute hash value, then find on the linked list
- Note that there are potentially as many as `table_size` lists



Why Lists?

- Can use **List ADT** for `find()`, `insert()`, and `delete()`
 - $O(l)$ running time where l is the number of elements in the particular chain
- Can also use **Binary Search Trees**
 - $O(\log l)$ time instead of $O(l)$
 - But the number of elements to search through should be **small**
 - Generally **not worth** the overhead of BSTs

Example

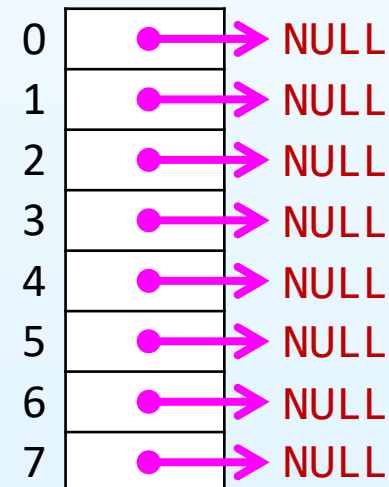
Let's store words into a table and allow a fast look-up

Design: We will store strings and the hash value of a string will be the **last 3 bits of the first character** in the word

– E.g., the hash of “optimal” is based on ‘o’

a	01100 001	b	01100 010	c	01100 011	d	01100 100	e	01100 101
f	01100 110	g	01100 111	h	01101 000	i	01101 001	j	01101 010
k	01101 011	l	01101 100	m	01101 101	n	01101 110	o	01101 111
p	01110 000	q	01110 001	r	01110 010	s	01110 011	t	01110 100
u	01110 101	v	01110 110	w	01110 111	x	01111 000	y	01111 001
z	01111 010								

– We thus create an array of **8 pointers**



Example: Hash Function

Let's store words into a table and allow a fast look-up

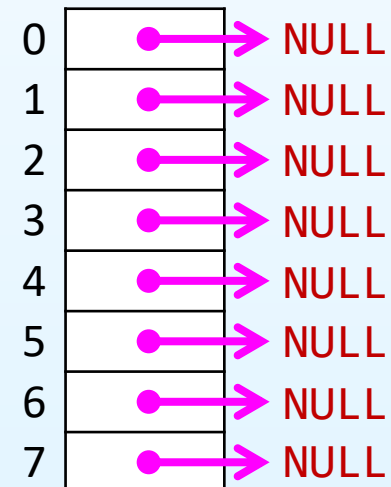
Design: Our hash function is

```
unsigned int hash(char *str) {  
    // the empty string "" is hashed to 0  
    if (strlen(str) == 0)  
        return 0;  
    return str[0] & 7;  
}
```

Bitwise operator (&)

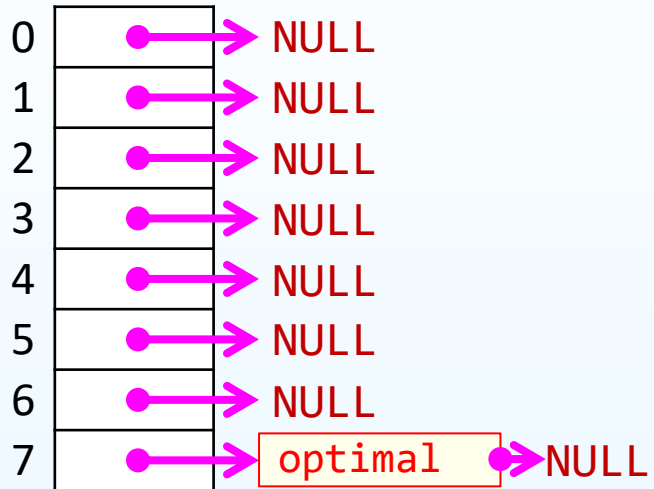
'a': 01100001
7: 00000111

 00000001



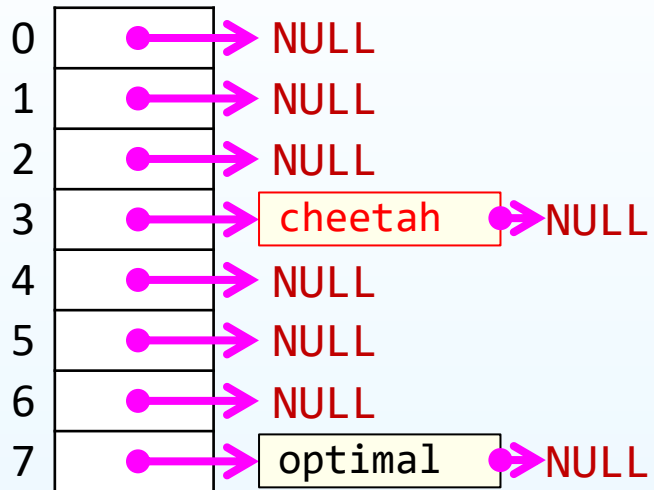
Example: insert()

- To do insert("optimal"), the first character 'o' is entered into the bin $01101111 = 7$



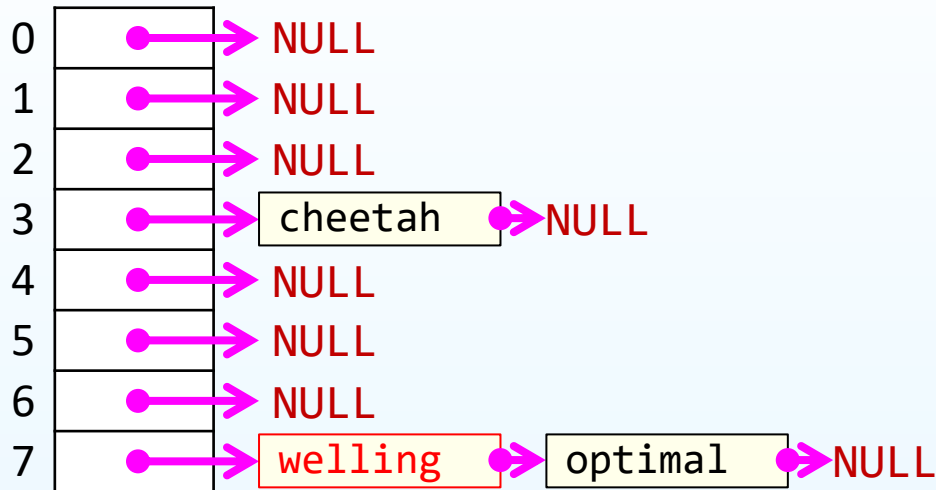
Example: insert()

- Similarly, insert("cheetah") will enter the word into the bin $01100011 = 3$



Example: insert()

- However, insert("welling") will enter the word into the same bin
 $01110111 = 7$

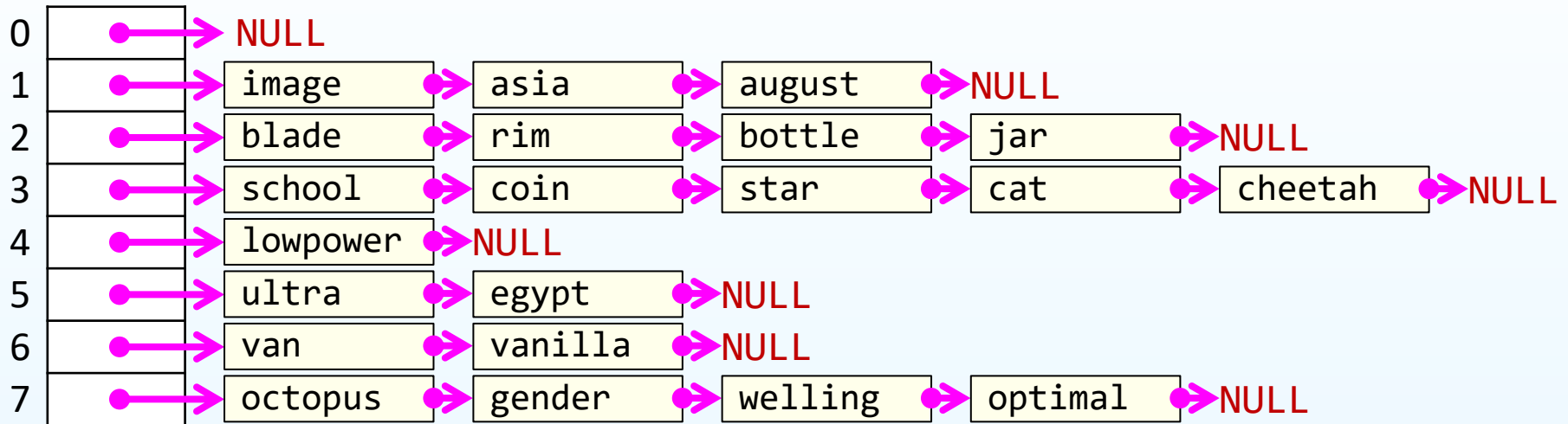


We will **heuristically** insert it at **front** of the linked list, **why?**

“The word accessed recently may be accessed again in the near future”

Example: insert()

- After 21 insertions, the linked lists are becoming **rather long**
 - We look for $\Theta(1)$ time, but must pay $O(l)$ for a linked list with l objects



Load Factor

- To describe the length of the linked lists, we define the load factor of the hash table:

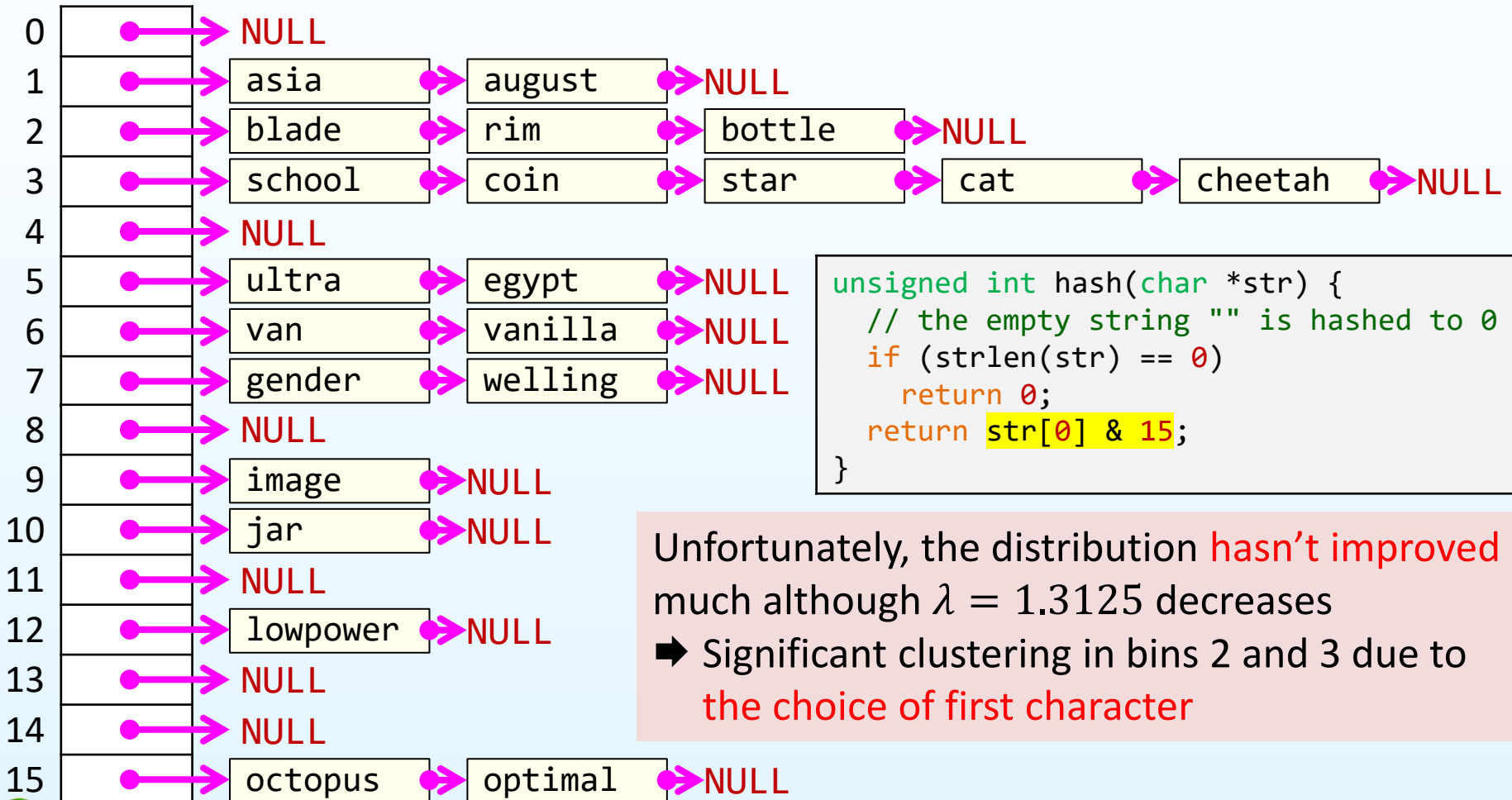
$$\lambda = \frac{n}{M}$$

This means the **average number of objects per bin**

- Right now, the previous insertions:
 - Each bin has $\lambda = \frac{21}{8} = 2.625$ objects on average
- If the load factor becomes too large, access time will increase to $\Theta(\lambda)$

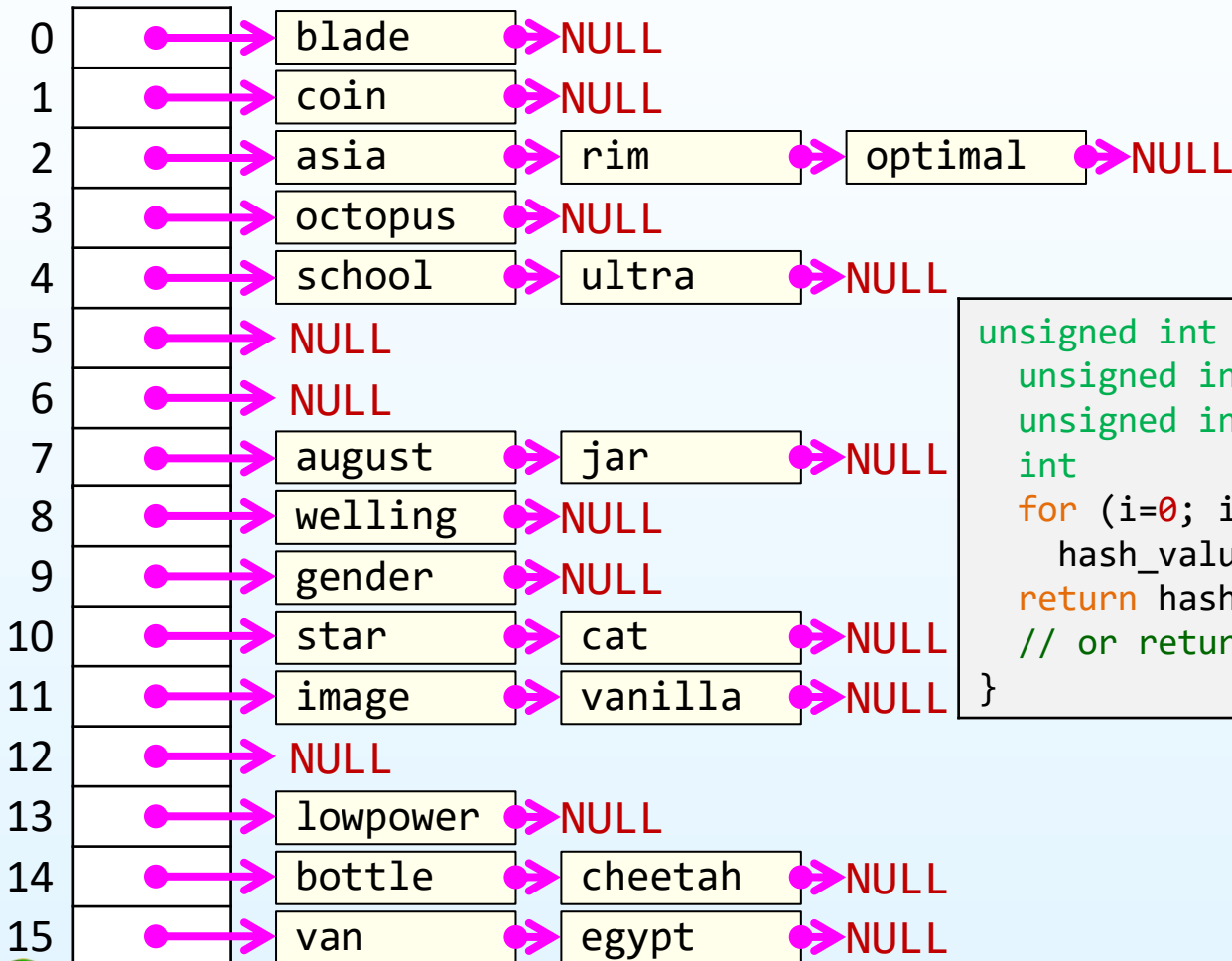
Example: Doubling Size

Design: We **double** the table size by changing the hash function to use the **last 4 bits** of the first character



Example: Choose a Good Hash Function

Design: Let's go back to the hash function defined previously



```
unsigned int hash(char *str) {  
    unsigned int prime = 12347;  
    unsigned int hash_value = 0;  
    int i = 0;  
    for (i=0; i<strlen(str); i++)  
        hash_value = prime*hash_value + str[i];  
    return hash_value % 16;  
    // or return hash_value & 15;  
}
```


Problems with Linked Lists

- One significant issue with chained hash tables using linked list
 - It requires **extra memory**
 - It uses **dynamic memory allocation**

Outline

- Hash ADT
- Hash Functions
- Collisions
 - Separate Chaining
 - Open Addressing

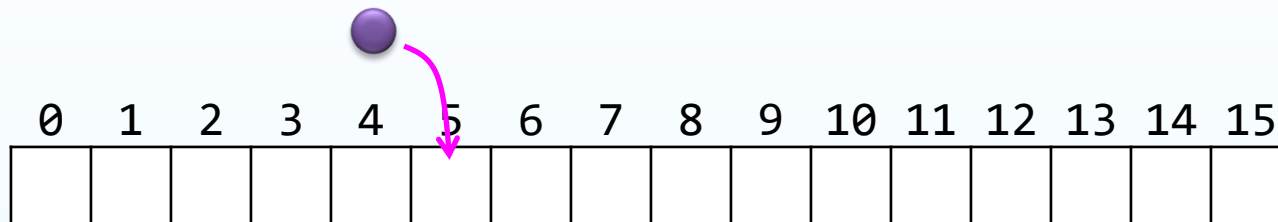
2. Resolution by Open Addressing

- **No links**, all object keys are in the table
 - Reduced overhead **saves space**
- We will define an **implicit rule** which tells us **where to look next**
 - When **inserting** x , we will look for the **first empty location** by $h_1(x), h_2(x), h_3(x), \dots$
 - When **searching** for x , check locations $h_1(x), h_2(x), h_3(x), \dots$ until either
 - x is **found**, or
 - we find an **empty location** (i.e., x is **not present**)
 - Various flavors of open addressing differ in which **probe sequence** they use

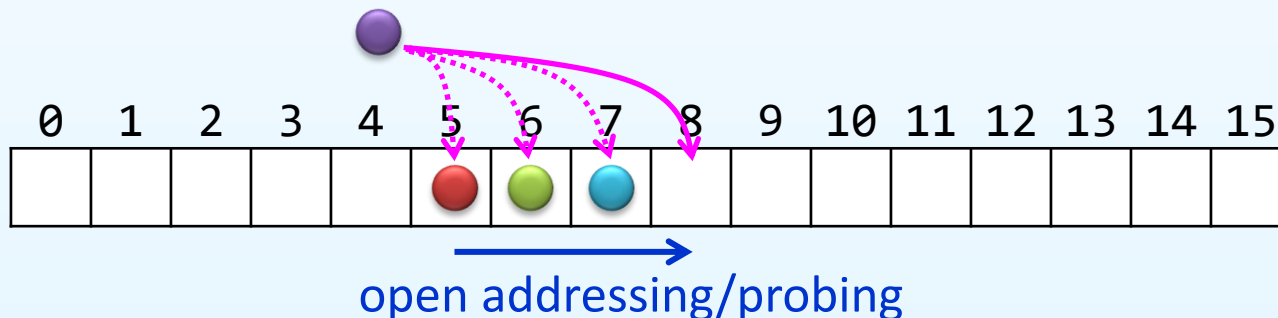
Open Addressing

Suppose an object hashes to bin 5

- If bin 5 is **empty**, we can insert the object into that entry



- **Otherwise**, we use an **implicit rule** to look for other **unoccupied** entry
 - **Rule:** continue searching until the first empty bin is found



- However, we can only **store as many objects as** there are entries in the array, i.e., **the load factor $\lambda < 1$**

Open Addressing

- The implicit rule is then defined as:

$$h_i(x) = (\text{hash}(x) + F(i)) \bmod \text{table_size}$$

for $i = 0, 1, 2, 3, \dots$ and $F(0) = 0$

- $F(i)$ is the **collision resolution function**; some possibilities:
 - **Linear probing**: $F(i) = i$
 - **Quadratic probing**: $F(i) = i^2$
 - **Double hashing**: $F(i) = i \cdot \text{hash}_2(x)$

2.1 Linear Probing

- The easiest method to probe the bins of the hash table is to search forward **linearly**:

$$F(i) = i$$

- Assume we are inserting into bin k :
 - If bin k is empty, we immediately insert into that bin
 - Otherwise, check bin $k + 1$, $k + 2$, and so on, until an empty bin is found
 - If we reach the end of the array, we start at the front (bin 0)

Example: insert()

- Consider a hash table with $M = 16$ bins
- Given a 3-digit hexadecimal number:
 - The **least-significant digit** is the primary hash function
- Insert these numbers:

¹19A, ¹207, ¹3AD, ¹488, 5BA, 680, 74C, 826, 946, ACD, B32, C8B, DBE, E9C

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
							207	488		19A			3AD		

Example: insert()

- Consider a hash table with $M = 16$ bins
- Given a 3-digit hexadecimal number:
 - The **least-significant digit** is the primary hash function
- Insert these numbers:

¹19A, ¹207, ¹3AD, ¹488, ²5BA, 680, 74C, 826, 946, ACD, B32, C8B, DBE, E9C

$$h_1(5BA) = (A + 1) \% 16 = B$$

collision occurs

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
							207	488		19A	5BA		3AD		



Example: insert()

- Consider a hash table with $M = 16$ bins
- Given a 3-digit hexadecimal number:
 - The **least-significant digit** is the primary hash function
- Insert these numbers:

¹19A, ¹207, ¹3AD, ¹488, ²5BA, ¹680, ¹74C, ¹826, 946, ACD, B32, C8B, DBE, E9C

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680						826	207	488		19A	5BA	74C	3AD		

Example: insert()

- Consider a hash table with $M = 16$ bins
- Given a 3-digit hexadecimal number:
 - The **least-significant digit** is the primary hash function
- Insert these numbers:

¹19A, ¹207, ¹3AD, ¹488, ²5BA, ¹680, ¹74C, ¹826, ⁴946, ACD, B32, C8B, DBE, E9C

collision occurs


0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680						826	207	488	946	19A	5BA	74C	3AD		



Example: insert()

- Consider a hash table with $M = 16$ bins
- Given a 3-digit hexadecimal number:
 - The **least-significant digit** is the primary hash function
- Insert these numbers:

¹19^A, ¹20⁷, ¹3A^D, ¹48⁸, ²5B^A, ¹68⁰, ¹74^C, ¹82⁶, ⁴94⁶, ²ACD, B3², C8^B, DB^E, E9^C

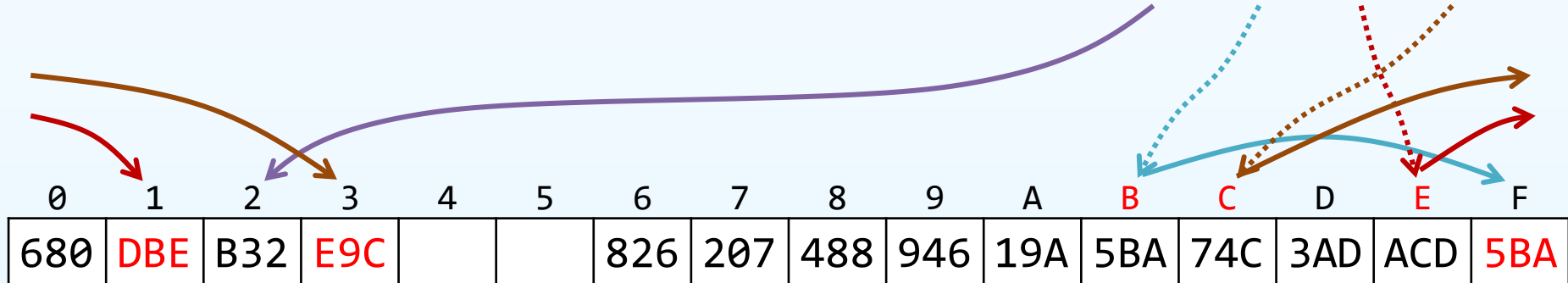


0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680						826	207	488	946	19A	5BA	74C	3AD	ACD	

Example: insert()

- Consider a hash table with $M = 16$ bins
- Given a 3-digit hexadecimal number:
 - The **least-significant digit** is the primary hash function
- Insert these numbers:

¹19A, ¹207, ¹3AD, ¹488, ²5BA, ¹680, ¹74C, ¹826, ⁴946, ²ACD, ¹B32, ⁵C8B, ⁴DBE, ⁸E9C



- We have completed all insertions:
 - The load factor is $\lambda = \frac{14}{16} = 0.875$
 - The average number of probes is $\frac{33}{14} \approx 2.36$

Example: Resizing the Array

- To **double** the capacity of the array, each value must be **rehashed**
 - We will use the **least-significant five bits** for hashing

```
unsigned int hash(int n) {
    return n & 31;
}
```

- Re-insert all previous numbers:

- After completing all insertions:

- The load factor is $\lambda = \frac{14}{32} = 0.4375$
- The average number of probes is $\frac{19}{14} \approx 1.36$

Example: find()

- Searching for membership is similar to insertions:
 - Start at the appropriate bin, and searching forward until
 - The item is **found**,
 - An empty bin is found ➡ **not found**, or
 - We have traversed the entire array ➡ **not found**
- Searching for 5C8

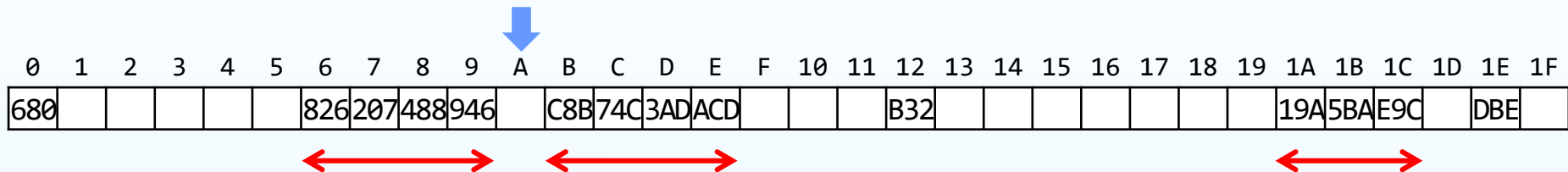
5C8 is not found

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
680						826	207	488	946	✗	C8B	74C	3AD	ACD				B32								19A	5BA	E9C		DBE	



Primary Clustering Phenomenon

- With more insertions, the **contiguous regions** (or **clusters**) get larger
 - ➔ This results in **longer search times**



- Above, we currently have three clusters
- There is a $\frac{5}{32} \approx 16\%$ chance that an insertion will fill bin A
 - Suppose 747 is inserted, two clusters will coalesce into one larger of length 9

2.2 Quadratic Probing

- Quadratic probing suggests moving forward by different amount:

$$F(i) = i^2$$

- Assume we are inserting into bin k :
 - If bin k is empty, we immediately insert into that bin
 - Otherwise, check bin $k + 1^2$, $k + 2^2$, and so on, until an empty bin is found
 - If we reach the end of the array, we start at the front (bin 0)

Will i^2 step all bins of the table?

```
1: #include <stdio.h>
2: unsigned int hash(unsigned int n, unsigned int M) {
3:     return n % M;
4: }
5: void try_quadratic(unsigned int n, unsigned int M) {
6:     unsigned int init = hash(n, M);
7:     unsigned int i;
8:     for (i=0; i<M; i++)
9:         printf("%3d", (init + i*i) % M);
10:    printf("\n");
11: }
12: int main(void) {
13:     try_quadratic(0, 10);
14:     try_quadratic(0, 16);
15:     return 0;
16: }
```

0 1 4 9 6 5 6 9 4 1

0 1 4 9 0 9 4 1 0 1 4 9 0 9 4 1



Making M Prime

```
12: int main(void) {  
13:     try_quadratic(0, 10);  
14:     try_quadratic(0, 11);  
15:     try_quadratic(0, 16);  
16:     try_quadratic(0, 17);  
17:     return 0;  
18: }
```

0	1	4	9	6	5	6	9	4	1							
0	1	4	9	5	3	3	5	9	4	1						
0	1	4	9	0	9	4	1	0	1	4	9	0	9	4	1	
0	1	4	9	16	8	2	15	13	13	15	2	8	16	9	4	1

Guarantee to iterate through $\left\lceil \frac{M}{2} \right\rceil$ entries

Problems:

- Modulus (%) is slow; cannot use bitwise operators (&, <<, or >>)
- Doubling the number of bins is difficult

Using the Powers of two $M = 2^m$

If $M = 2^m$ is powers of two, this guarantees that all M entries are visited:

$$F(i) = c_1 i + c_2 i^2$$

Since $i + i^2$ is always even, we can select $c_1 = c_2 = \frac{1}{2}$

Using $M = 2^m$

```
1: #include <stdio.h>
2: unsigned int hash(unsigned int n, unsigned int M) {
3:     return n % M;
4: }
5: void try_quadratic(unsigned int n, unsigned int M) {
6:     unsigned int init = hash(n, M);
7:     unsigned int i;
8:     for (i=0; i<M; i++)
9:         printf("%3d", (init + (i + i*i)/2) % M);
10:    printf("\n");
11: }
12: int main(void) {
13:     try_quadratic(0, 10); // not powers of two
14:     try_quadratic(0, 16); // powers of two (2^4)
15:     return 0;
16: }
```

0 1 3 6 0 5 1 8 6 5

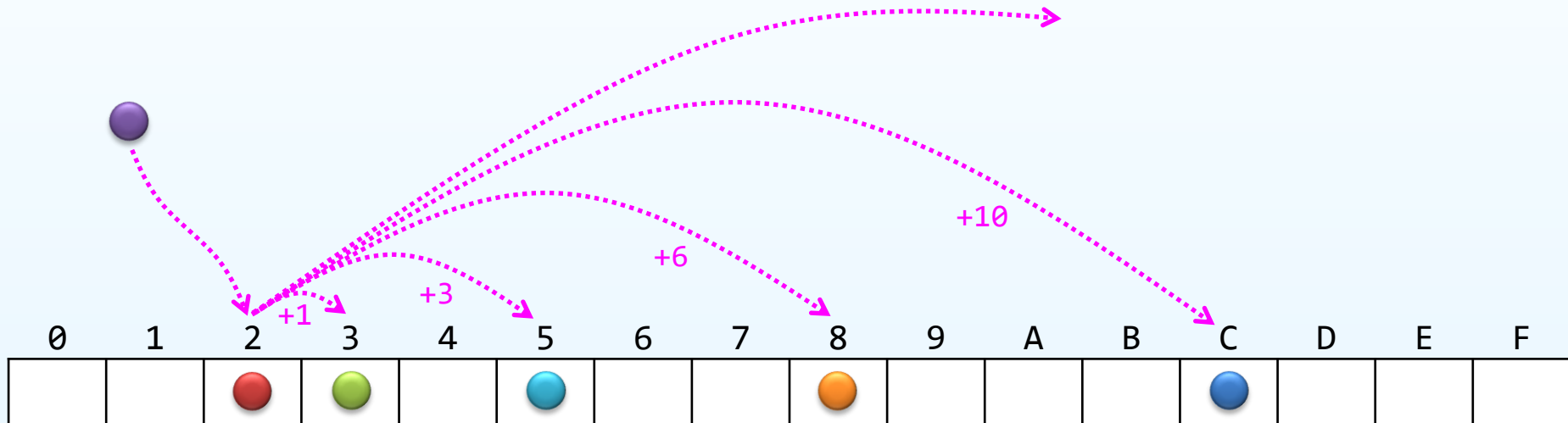
0 1 3 6 10 15 5 12 4 13 7 2 14 11 9 8



Example: insert() and delete()

- Consider a hash table with $M = 16$ bins
- Suppose try to insert at 2

$$h_i(x) = (2 + (i + i^2)/2) \% 16 \quad \text{for } i = 0, 1, 2, \dots$$



Secondary Clustering Phenomenon

- Weakness with quadratic problem:
 - It reverts to linear probing if many of the hash function **is not random**
 - Objects placed in the same bin will follow the **same sequence**

2.3 Double Hashing

- Double hashing suggests moving forward by a **jump function**:

$$F(i) = i \cdot hash_2(x)$$

- Assume we are inserting into bin k :
 - If bin k is empty, we immediately insert into that bin
 - Otherwise, check bin $k + hash_2(x)$, $k + 2 \cdot hash_2(x)$, and so on, until an empty bin is found
 - If we reach the end of the array, we start at the front (bin 0)

Example of Double Hashing

```
1: #include <stdio.h>
2: #define PRIME 7
3: unsigned int hash(unsigned int n, unsigned int M) {
4:     return n % M;
5: }
6: void try_doublehash(unsigned int n, unsigned int M) {
7:     unsigned int init = hash(n, M);
8:     unsigned int jump = PRIME - hash(n, PRIME);
9:     unsigned int i;
10:    for (i=0; i<M; i++)
11:        printf("%3d", (init + i*jump) % M);
12:    printf("\n");
13: }
14: int main(void) {
15:     try_doublehash(0, 10);
16:     try_doublehash(0, 16);
17:     return 0;
18: }
```

0 7 4 1 8 5 2 9 6 3

0 7 14 5 12 3 10 1 8 15 6 13 4 11 2 9



Be Careful !!!

Must be careful about **the second hash function** $hash_2(x)$:

$$h_i(x) = (hash_1(x) + i \cdot hash_2(x)) \bmod table_size$$

for $i = 0, 1, 2, 3, \dots$

- **Must never evaluate to zero**
 - ➡ Otherwise, it will always hash the same value
- **Must make sure that all bins can be probed**
 - E.g., if $hash_1(x) = 3$, $hash_2(x) = 9$, and $M = 12$,
then we can probe only the bins $(3 + 9)\%12 = 0$,
 $(3 + 18)\%12 = 9$, $(3 + 27)\%12 = 6$, $(3 + 36)\%12 = 3$,
 $(3 + 45)\%12 = 0$, and so on

Any Question?

