Lecture 6

COMP 3760

Space/time trade-offs

Text chapter 7

Space/time tradeoffs

- **Space** refers to the memory consumed by an algorithm to complete its execution
- **Time** refers to the required time for an algorithm to complete the execution
- The best algorithm is one that
  - Requires less memory
  - AND takes less time

In practice this is not always possible

Space/time tradeoffs

- We have to sacrifice one at the cost of the other.

- If space is our constraint, then we have to choose an algorithm that requires less space at the cost of more execution time. (example: Bubble Sort, Selection Sort)

- If time is our constraint then we have to choose an algorithm that takes less time to complete its execution at the cost of more space. (example: MergeSort)

Types of space/time tradeoffs

1. **<u>Input enhancement:</u>** preprocess the input to store some info to be used later in solving the problem
   - Comparison Counting Sort
   - Distribution Counting Sort
   - String Matching (improved algorithm)

2. **<u>Pre-structuring:</u>** use extra space to facilitate faster access to the data
   - Hashing
   - Hash Function
   - Collision Handling
   - Efficiency of Hashing

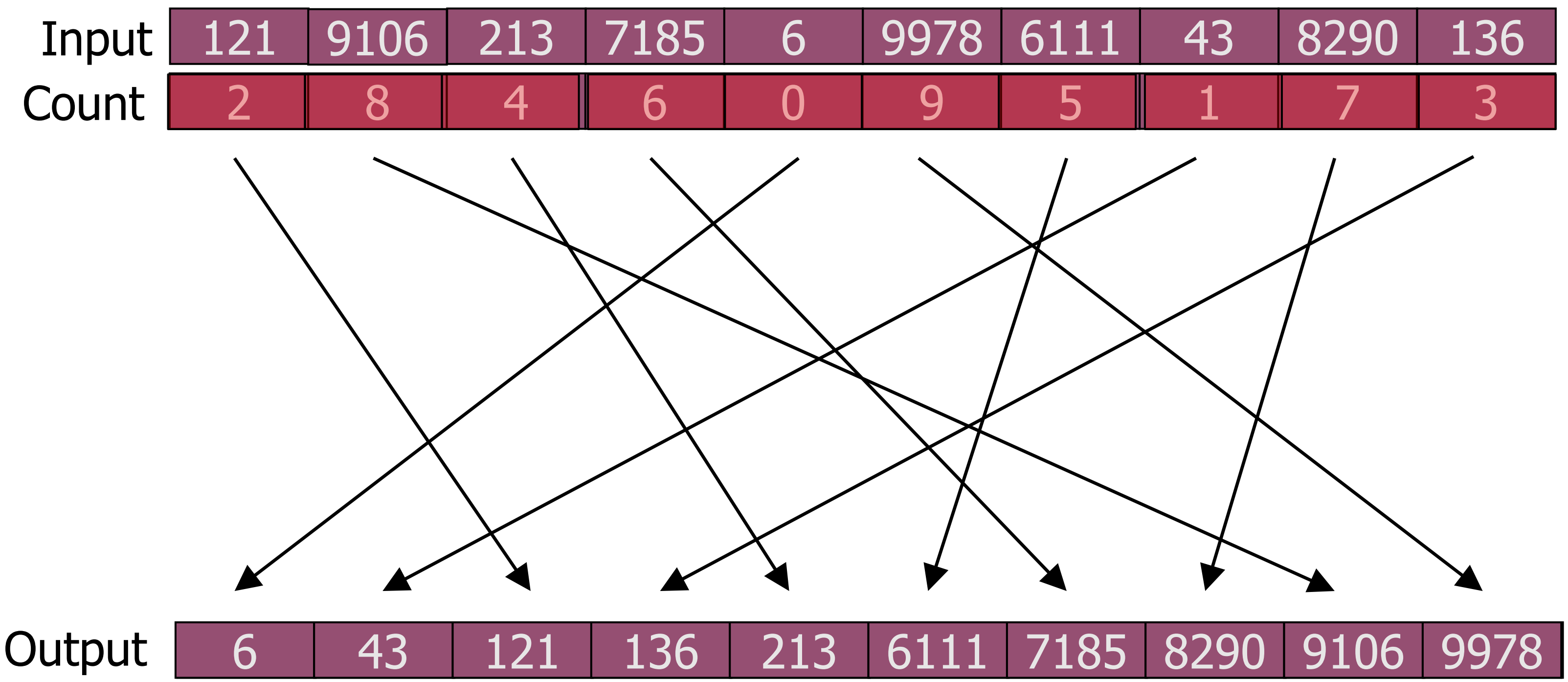
Comparison Counting Sort

- Idea: for each element of a list to be sorted, count the total number of elements smaller than this element and record the results in a table.

Input	121	9106	213	7185	6	9978	6111	43	8290	136
Count	2	8	4	6	0	9	5	1	7	3

Comparison Counting Sort

- Now move each input element to its corresponding position



Comparison Counting Sort

```
Algorithm ComparisonCountingSort(A[0..n-1])
  for i ← 0 to n-2
     for j ← i+1 to n-1
        if input[i] < input[j]
           Count[j]++
        else
           Count[i]++
  for i ← 0 to n-1
     output[Count[i]] ← input[i]
```

Efficiency of CCS

- It's $O(n^2)$
  - Of course we know a couple of algorithms that are $O(n \log n)$: MergeSort and HeapSort

Types of space/time tradeoffs

1. **Input enhancement:** preprocess the input to store some info to be used later in solving the problem
    - Comparison Counting Sort
    - Distribution Counting Sort
    - String Matching

2. **Pre-structuring:** uses extra space to facilitate faster access to the data.
    - Hashing
    - Hash Function
    - Collision Handling
    - Efficiency of Hashing

Distribution Counting Sort

- Suppose we need to sort an array with a “small” set of known values

8	5	7	6	7	8	5	8	6	8	8	5

Distribution Counting Sort

- Idea: count how many of each number...

8	5	7	6	7	8	5	8	6	8	8	5

- ...and determine the distribution from that
  - three 5's → positions 0 to 2
  - two 6's → positions 3 to 4
  - two 7's → positions 5 to 6
  - five 8's → positions 7 to 11 (11 is n-1)

5	5	5	6	6	7	7	8	8	8	8	8

Distribution Counting Sort

Algo DistributionCountingSort (A[0.. n-1])

for $j \leftarrow 0$ **to** $u\text{-}l$ **do**

$\qquad C[j] \leftarrow 0$

for $i \leftarrow 0$ **to** $n\text{-}1$ **do**

$\qquad C[A[i]\text{-}l] \leftarrow C[A[i]\text{-}l] + 1$

for $j \leftarrow 1$ **to** $u\text{-}l$ **do**

$\qquad C[j] \leftarrow C[j\text{-}1] + C[j]$

for $i \leftarrow n\text{-}1$ **downto** 0 **do**

$\qquad j \leftarrow \mathrm{A}[i]\text{-}\ l$

$\qquad S[C[j]\ \text{-}1] \leftarrow \mathrm{A}[i]$

$\qquad C[j] \leftarrow C[j] - 1$

return S

Distribution Counting Sort-example

u:14

l:11

A:

14	11	13	14	13

Size: u - l+1 = k

#of11	#of12	#of13	#of14

C:

One "bucket" for each different value we might encounter

S:

This will be the sorted array

Loop 1: initialization

A:

14	11	13	14	13

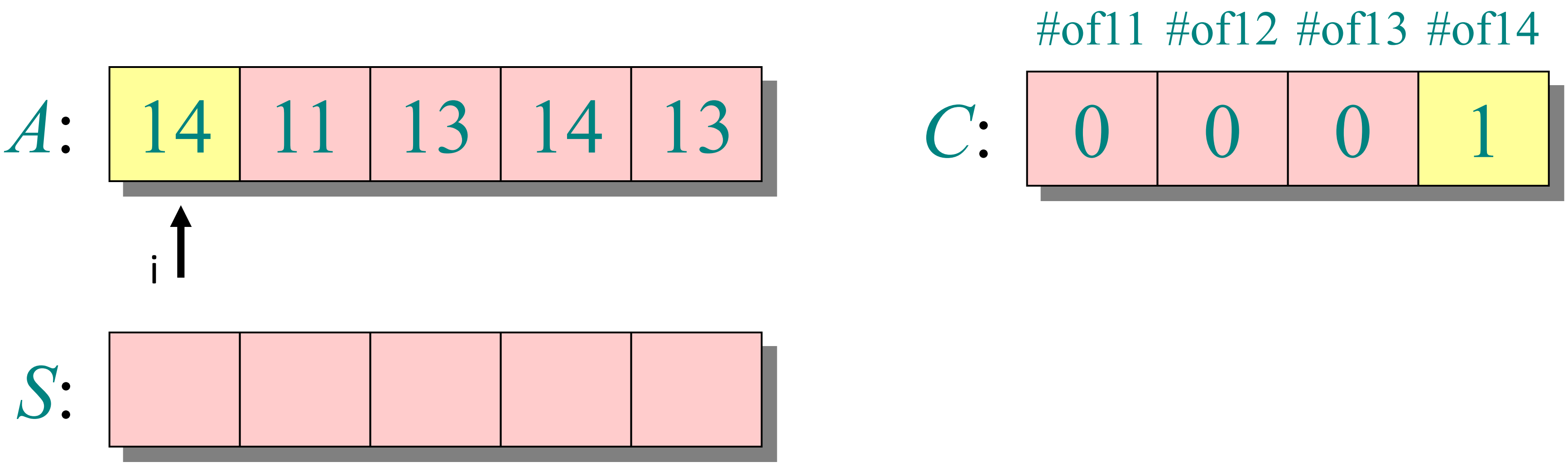
C:

#of11	#of12	#of13	#of14
0	0	0	0

S:

1. **for** $j \leftarrow 0$ **to** u-l **do**
   $C[j] \leftarrow 0$

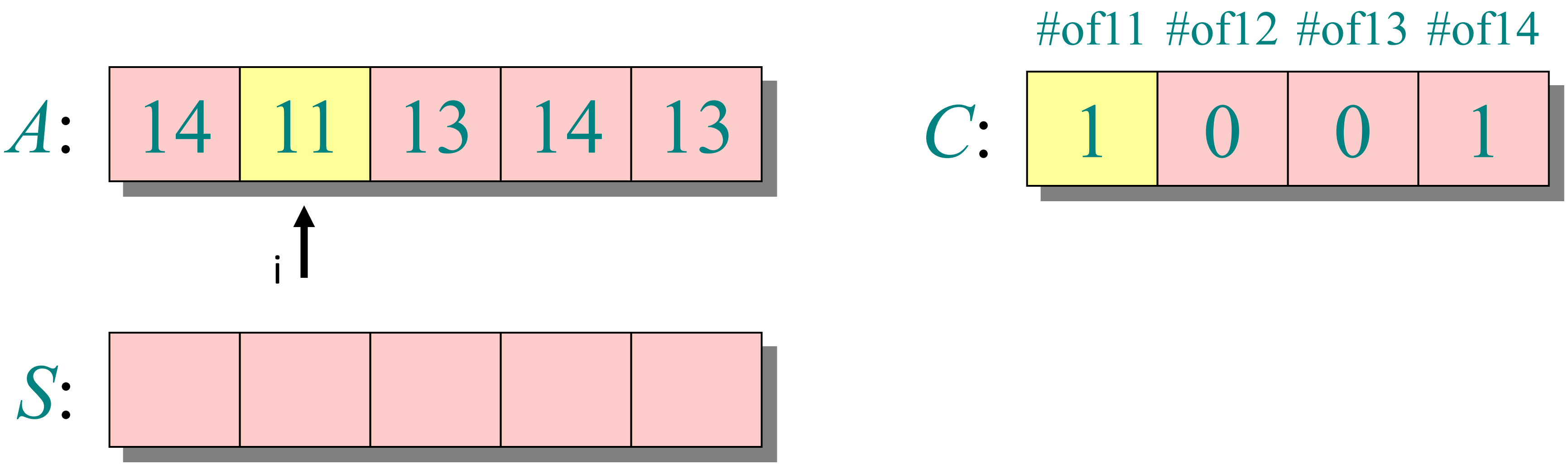
Loop 2: count



2. **for** $i \leftarrow 0$ **to** $n\text{-}1$ **do**

$C[A[\,i]\text{-}l] \leftarrow C[A[\,i]\text{-}l] + 1$

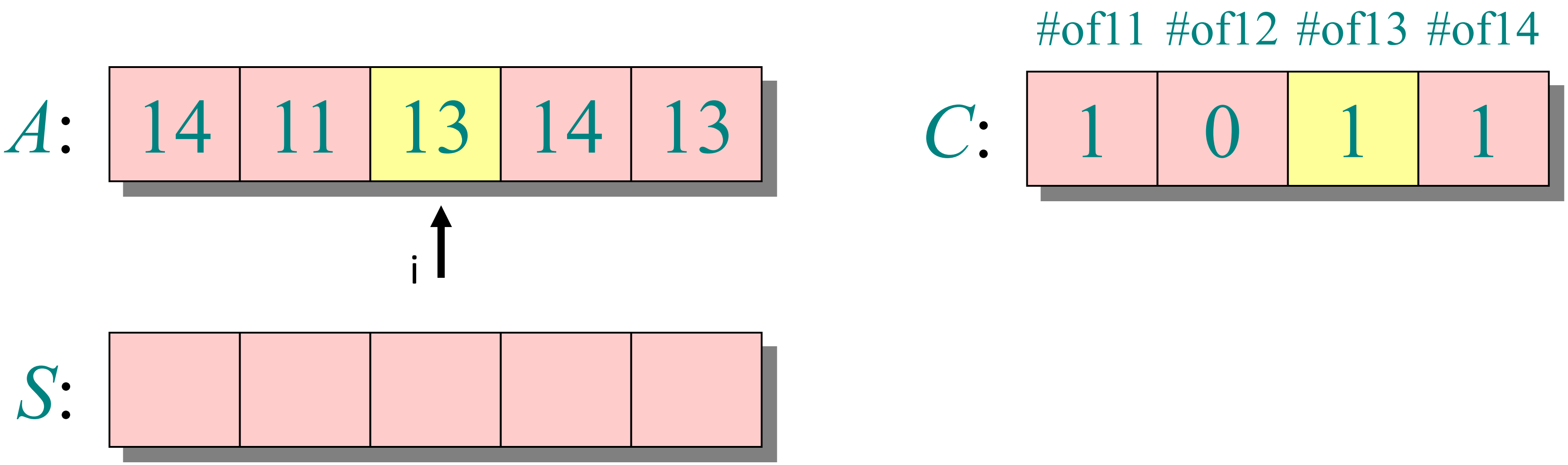
Loop 2: count



2. **for** $i \leftarrow 0$ **to** $n\text{-}1$ **do**

$$C[A[\,i]-l] \leftarrow C[A[\,i]-l] + 1$$

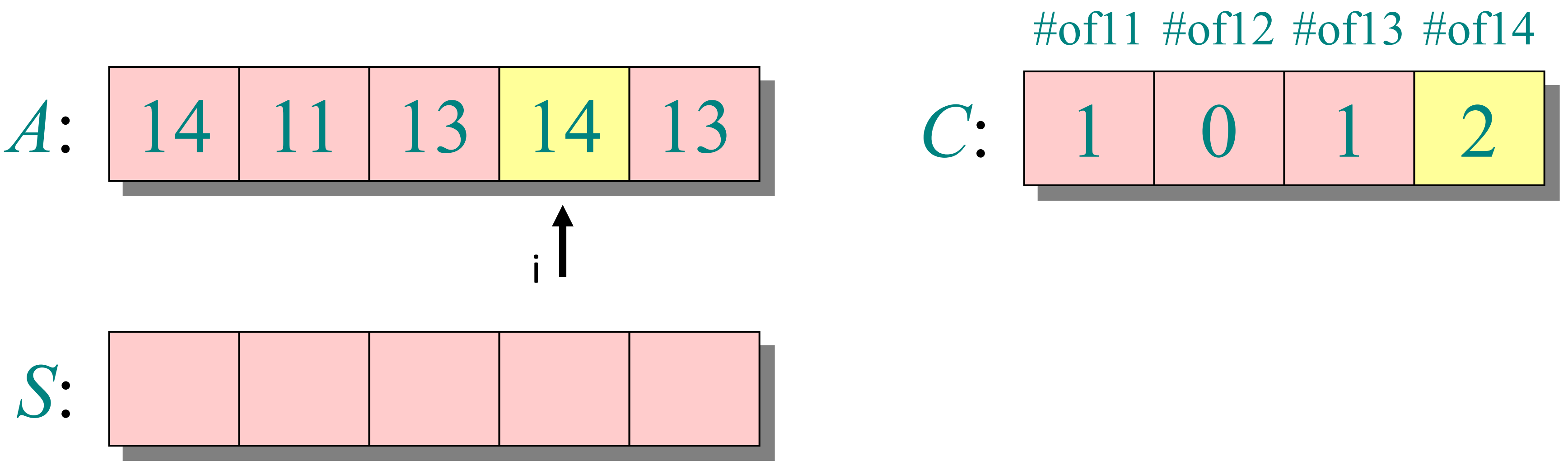
Loop 2: count



2. for $i \leftarrow 0$ **to** $n\text{-}l$ **do**

$$C[A[\,i]\text{-}l] \leftarrow C[A[\,i]\text{-}l] + 1$$

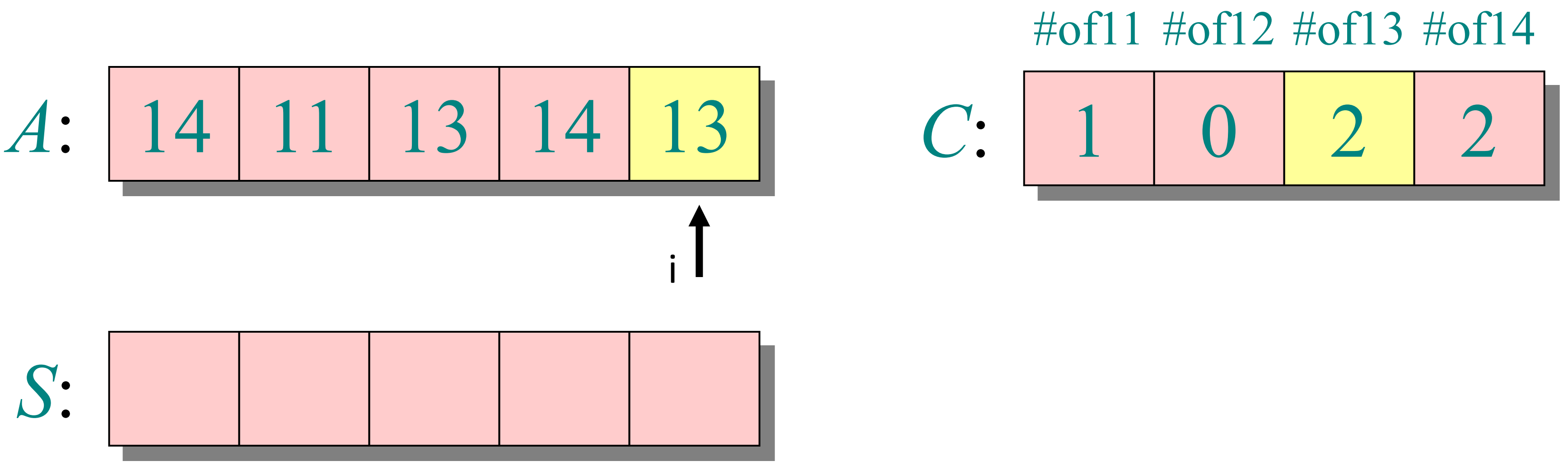
Loop 2: count



2. for $i \leftarrow 0$ **to** $n\text{-}l$ **do**

$$C[A[\,i]\text{-}l] \leftarrow C[A[\,i]\text{-}l] + 1$$

Loop 2: count



2. **for** $i \leftarrow 0$ **to** $n\text{-}1$ **do**

$C[A[i]\text{-}l] \leftarrow C[A[i]\text{-}l] + 1$

Loop 3: compute running sum

A:

14	11	13	14	13

S:

	#of11	#of12	#of13	#of14
C:	1	0	2	2

j

C:	1	1	2	2
		#of 11+12		

3. for $j \leftarrow 1$ **to** $u\text{-}l$ **do**

$C[j] \leftarrow C[j\text{-}1] + C[j]$

Loop 3: compute running sum

A:

14	11	13	14	13

S:

C:

#of11	#of 11+12	#of13	#of14
1	1	2	2

j

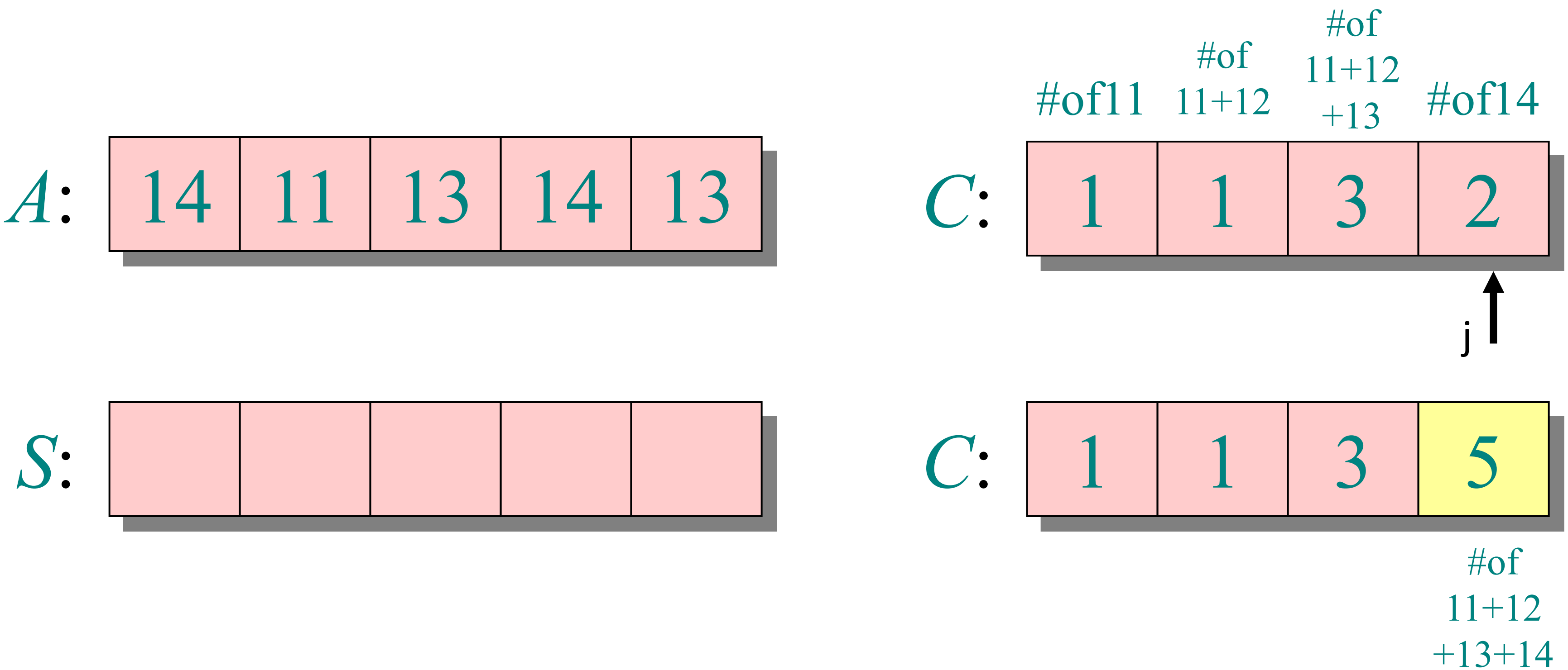
C:

1	1	3	2
		#of 11+12 +13	

3. for $j \leftarrow 1$ **to** $u\text{-}l$ **do**

$C[j] \leftarrow C[j\text{-}1] + C[j]$

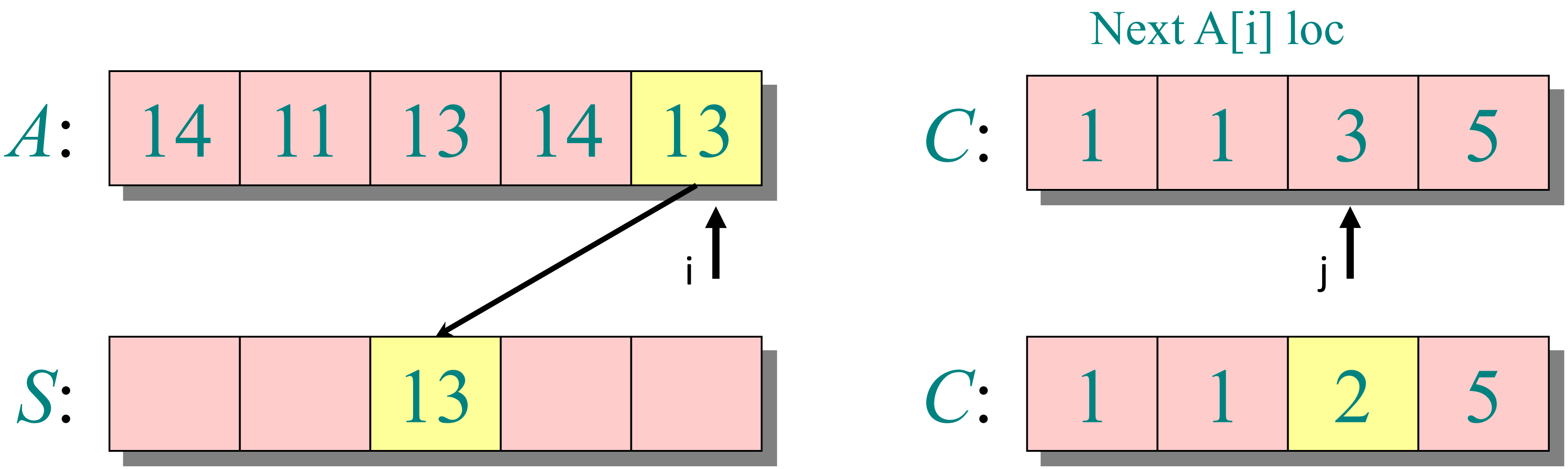
Loop 3: compute running sum



3. **for** $j \leftarrow 1$ **to** $u\text{-}l$ **do**

$C[j] \leftarrow C[j\text{-}1] + C[j]$

Loop 4: re-arrange



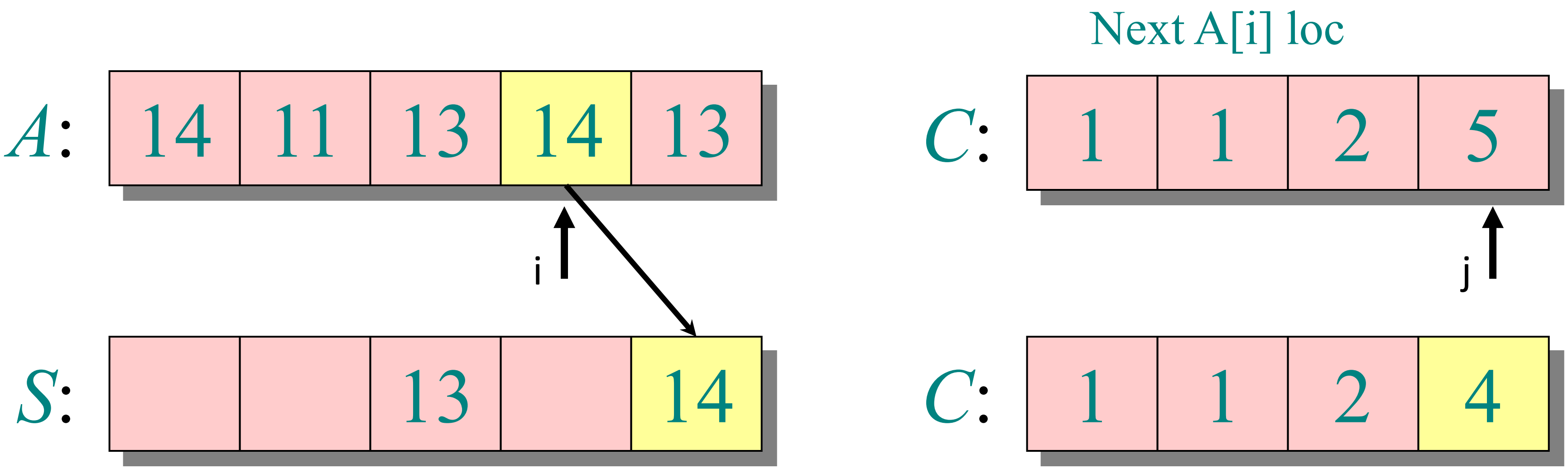
4. for $i \leftarrow n\text{-}1$ **downto** 0 **do**

$j \leftarrow \mathrm{A}[i] - l$

$S[C[j] - 1] \leftarrow \mathrm{A}[i]$

$C[j] \leftarrow C[j] - 1$

Loop 4: re-arrange



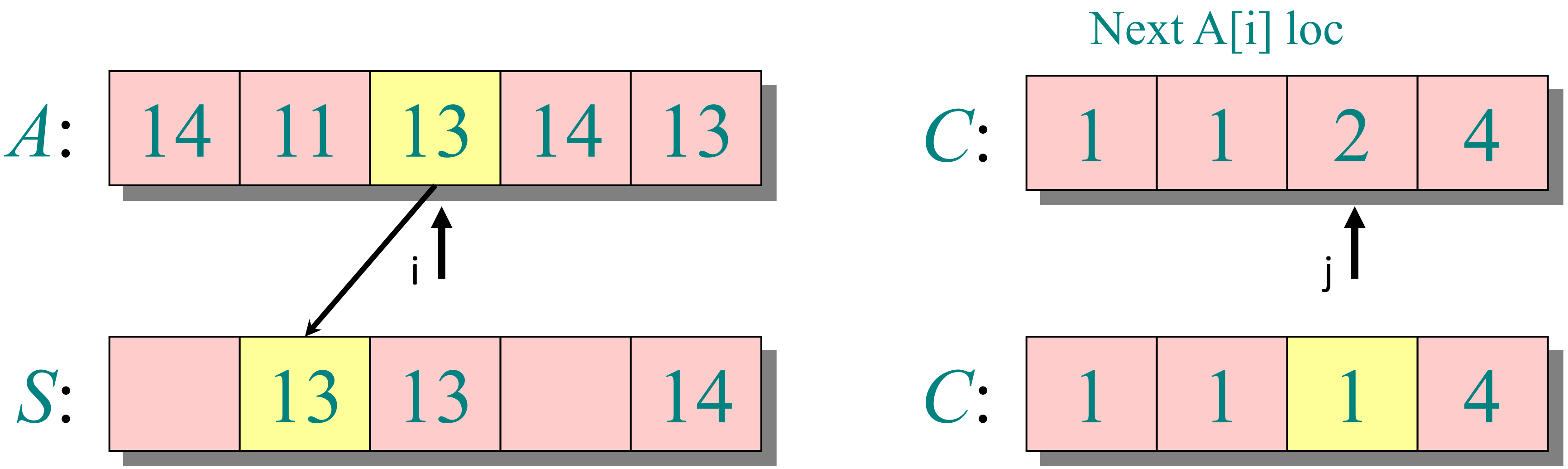
4. **for** $i \leftarrow n\text{-}1$ **downto** 0 **do**

$j \leftarrow \mathrm{A}[i] - l$

$S[C[j] - 1] \leftarrow \mathrm{A}[i]$

$C[j] \leftarrow C[j] - 1$

Loop 4: re-arrange



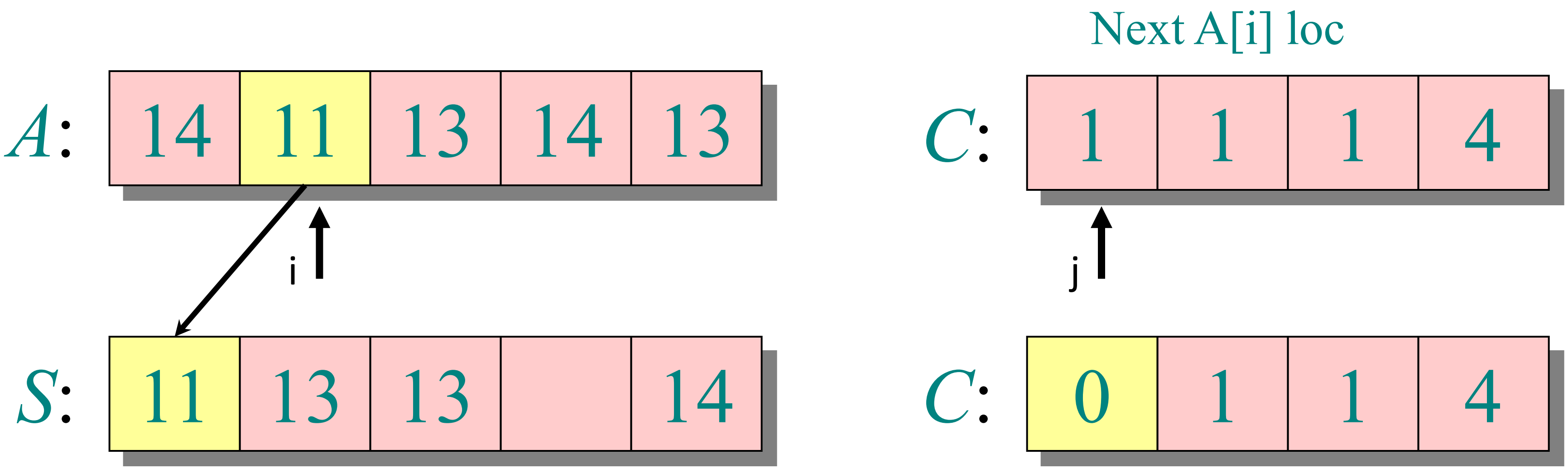
4. for $i \leftarrow n\text{-}1$ **downto** 0 **do**

$j \leftarrow \mathrm{A}[i] - l$

$S[C[j] - 1] \leftarrow \mathrm{A}[i]$

$C[j] \leftarrow C[j] - 1$

Loop 4: re-arrange



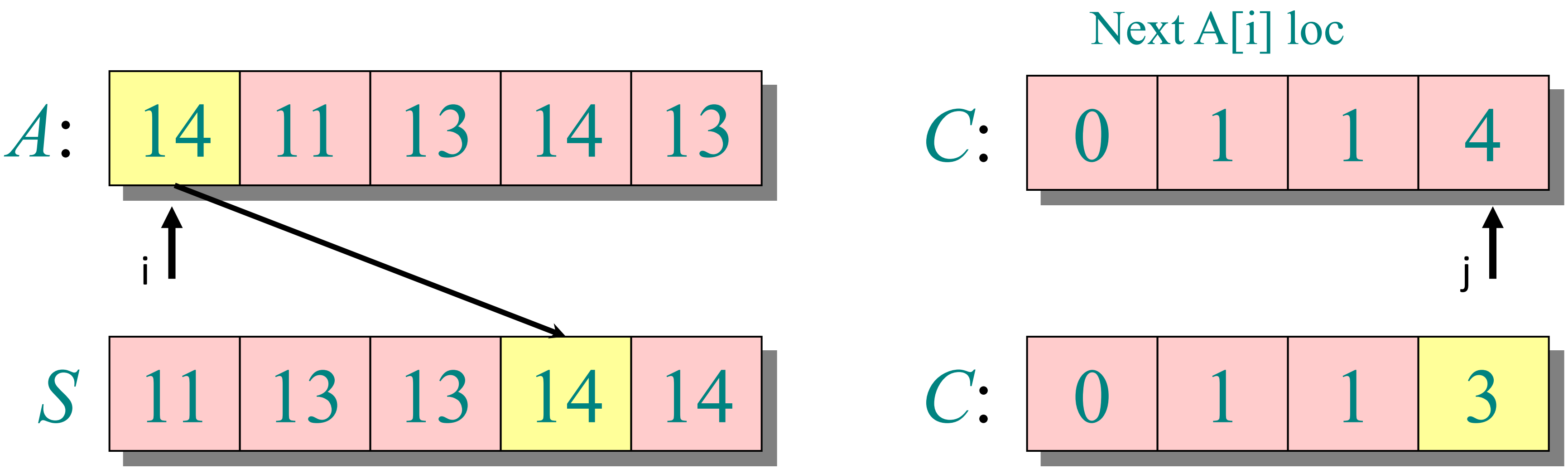
4. for $i \leftarrow n\text{-}1$ **downto** 0 **do**

$j \leftarrow \mathrm{A}[i] - l$

$S[C[j] - 1] \leftarrow \mathrm{A}[i]$

$C[j] \leftarrow C[j] - 1$

Loop 4: re-arrange



4. for $i \leftarrow n\text{-}1$ **downto** 0 **do**

$j \leftarrow \mathrm{A}[i] - l$

$S[C[j] - 1] \leftarrow \mathrm{A}[i]$

$C[j] \leftarrow C[j] - 1$

Algo DistributionCountingSort (A[0.. n-1])

$O(k)$

for $j \leftarrow 0$ **to** $u\text{-}l$ **do**

  $C[j] \leftarrow 0$

$O(n)$

for $i \leftarrow 0$ **to** $n\text{-}1$ **do**

  $C[A[i]\text{-}l] \leftarrow C[A[i]\text{-}l] + 1$

$O(k)$

for $j \leftarrow 1$ **to** $u\text{-}l$ **do**

  $C[j] \leftarrow C[j\text{-}1] + C[j]$

$O(n)$

for $i \leftarrow n\text{-}1$ **downto** 0 **do**

  $j \leftarrow \mathrm{A}[i]\text{-}\ l$

  $S[C[j]\ \text{-}1] \leftarrow \mathrm{A}[i]$

  $C[j] \leftarrow C[j] - 1$

$O(n + k)$

return S

Efficiency of Distribution Counting Sort

- If the range of input values is roughly <= the number of input values
  - … then this algorithm is O(n)

- This is really, really good!
  - But it is a *special-purpose* algorithm
  - Significant constraint on the *range* of input values

Types of space/time tradeoffs

1. **Input enhancement:** preprocess the input to store some info to be used later in solving the problem
    - Comparison Counting Sort
    - Distribution Counting Sort
    - String Matching

2. **Pre-structuring:** uses extra space to facilitate faster access to the data.
    - Hashing
    - Hash Function
    - Collision Handling
    - Efficiency of Hashing

String Matching: reminder

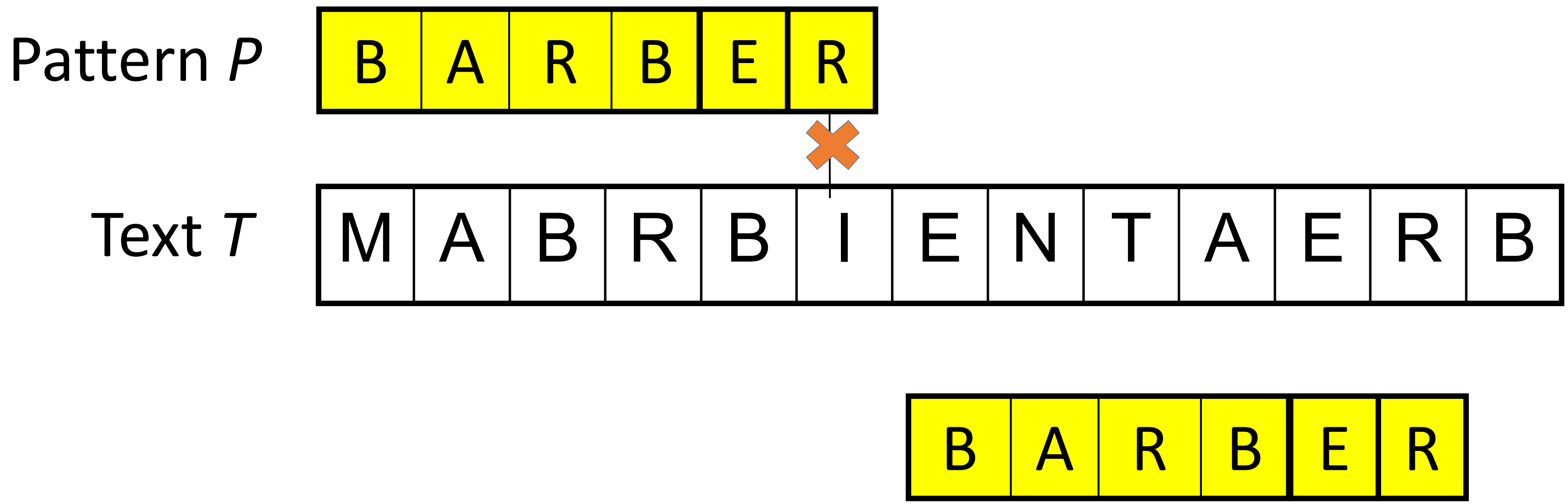
- Pattern: a string of m characters to search for
- Text: a (long) string of n characters to search in
- Brute force algorithm:
  - Align pattern at beginning of text
  - Moving L-R within pattern, compare pattern to text until
    - All characters are found to match (successful search); or
    - A mismatch is detected
  - While pattern is not found and the text is not yet exhausted, shift pattern one position to the right and repeat step 2.
- Time Complexity: $O((n-m+1) \times m)$

Input Enhancement in String Matching

- How can we improve string matching by using the concept of *input enhancement*?

- Useful observation: Whenever we have a mismatch, maybe we can shift the pattern over by *more than one character* before comparing again

Input Enhancement in String Matching



- *Compare the chars **right to left***
- *There is no "I" in BARBER, so we should shift the pattern all the way past the "I"*
- Determine the number of shifts by looking at the character of the text that is aligned against the last character of the pattern

String Matching: Key Observation

- Consider, as an example, searching for the pattern BARBER in some text. Here is *a moment in time*:

$$\begin{array}{ccccccccccc} s_0 & \cdots & & & & & c & \cdots & s_{n-1} \\ & & \mathrm{B} & \mathrm{A} & \mathrm{R} & \mathrm{B} & \mathrm{E} & \mathrm{R} & \end{array}$$

- *When a mismatch occurs, look at the Text character that is aligned with the rightmost character of P*

String Matching: Four cases

- Text char *c* never appears in the Pattern
- Text char *c* appears in the Pattern but *not last*
- Text char *c* appears last in the Pattern but *only that one time*
- Text char *c* appears last in the Pattern *and other times*

String Matching: Four cases

- Case 1: If the Text char *c* never appears in the Pattern...

$s_0 \quad \ldots \quad S \quad \ldots \quad s_{n-1}$

B A R B E R

B A R B E R

...we can shift Pattern by its entire length

String Matching: Four cases

- Case 2: If the Text char *c* appears in the Pattern but *not last*...

$s_0 \quad \cdots \qquad \qquad \qquad \qquad \mathrm{B} \qquad \qquad \qquad \qquad \cdots \quad s_{n-1}$

$\nparallel$

B A R B E R

B A R B E R

...we can shift to align the last occurrence of *c* in Pattern with *c* in Text

String Matching: Four cases

- Case 3: If the Text char *c* appears last in the Pattern but *only that one time*...

```
s_0   . . .          M  E  R                        . . .   s_{n-1}
                     ≠  ‖  ‖
             L  E  A  D  E  R
                             L  E  A  D  E  R
```

...we can shift Pattern by its entire length

String Matching: Four cases

- Case 4: If the Text char *c* appears last in the Pattern *and other times*...

s_0 . . . A R . . . s_{n-1}

$\not\parallel$ $\parallel$

R E O R D E R

R E O R D E R

...we can shift to align the second-to-last occurrence of *c* in Pattern with *c* in Text

The Strategy

- How can we use this observation for input enhancement?

- Strategy:
  - Create a “shift table”
    - One entry for each possible value in the *input alphabet*
  - Shift table will indicate the number of positions to shift the pattern

Table	**2**	**5**	**7**	**2**	**7**	**7**	**3**	**…**	**7**	**4**	**7**
	0	**1**	**2**	**3**	**4**	**5**	**6**		**23**	**24**	**25**
	“A”	**“B”**	**“C”**	**“D”**	**“E”**	**“F”**	**“G”**		**“X”**	**“Y”**	**“Z”**

The Shift Table

- How to construct the shift table:
  - it will have a size equal to the number of elements in the input alphabet (so we have to know this in advance!)

$$t(c) = \begin{cases} \text{distance from } c\text{'s rightmost occurrence in pattern} \\ \text{among its first } m\text{-1 characters to its right end} \\ \\ \text{pattern's length } m\text{, otherwise} \end{cases}$$

The Shift Table

- Example:
  - assume our alphabet is {A B C D E F G H I J}
  - assume our pattern is IDIGDAB (m=7)

  - When a mismatch occurs, what character is aligned with our pattern?

... text ... text ... text ... text ... text ... text ... text ... text ... text ... text ... text ...

I D I G D A B

Table	**1**	**7**	**7**	**2**	**7**	**7**	**3**	**7**	**4**	**7**
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
	"A"	**"B"**	**"C"**	**"D"**	**"E"**	**"F"**	**"G"**	**"H"**	**"I"**	**"J"**

Using the shift table …

- Example:

there is a mismatch on the first compare, so we lookup **table["D"]**, which returns **2**, so we shift by 2 …

Pattern *P*

I	D	I	G	D	A	B

text *T*

I	B	A	G	H	J	D	A	B	A	D	A	B

I	D	I	G	D	A	B

Table

1	**7**	**7**	**2**	**7**	**7**	**3**	**7**	**4**	**7**
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
"A"	**"B"**	**"C"**	**"D"**	**"E"**	**"F"**	**"G"**	**"H"**	**"I"**	**"J"**

Using the shift table …

- Example:

there is a mismatch, so we lookup **table[“B”]**, which returns **7**, so we shift by 7 .

text *T*	I	B	A	G	H	J	D	A	B	A	D	A	B
			I	D	I	G	D	A	B				

Table	**1**	**7**	**7**	**2**	**7**	**7**	**3**	**7**	**4**	**7**
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
	“A”	**“B”**	**“C”**	**“D”**	**“E”**	**“F”**	**“G”**	**“H”**	**“I”**	**“J”**

More details about this algorithm in the textbook.

(it is called Horspool's algorithm)

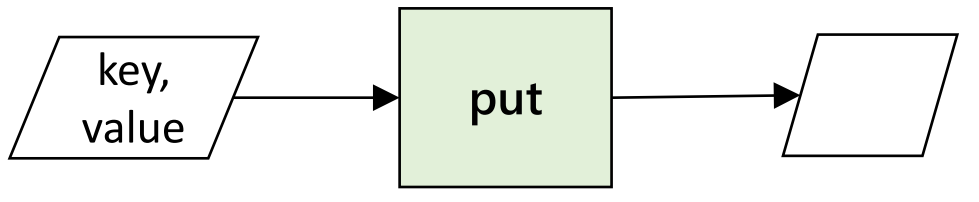
Types of space/time tradeoffs

1. **Input enhancement:** preprocess the input to store some info to be used later in solving the problem
    - Comparison Counting Sort
    - Distribution Counting Sort
    - String Matching

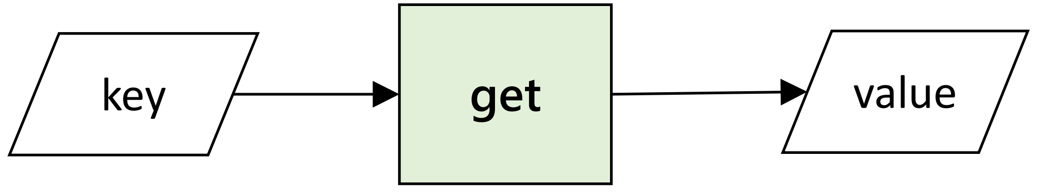
2. **Pre-structuring:** uses extra space to facilitate faster access to the data.
    - Hashing
    - Hash Function
    - Collision Handling
    - Efficiency of Hashing

You know about HashMaps

- Map.put(key, value)



- value = Map.get(key)



Today we are looking inside these boxes.

Fast Storage of Keyed Records

Goal: want some way to do fast storage/lookups/retrieval of information, based on an arbitrary key

eg:

```
key = A00043526
value = Jimmy
```

Let's consider traditional data structures ...

Array: How would you use an array (or arrays) to store this

- use either 2 1D arrays or 1 2D array or an array of objects
  - store key in a sorted array (for fast retrieve)
  - use the second array (or column) to store the record or a pointer to the record ... or ...
- alternatively, create an object 'Employee', and store in an array of objects

Using Sorted Array

Two 1D Arrays ...

1	A00043522
2	A00666666
3	
4	
	⋮
n-1	
n	

1	Jimmy
2	beelzebub
3	
4	
	⋮
n-1	
n	

One 2D Array ...

1	A00043522	Jimmy
2	A00666666	beelzebub
3		
4		
	⋮	⋮
n-1		
n		

Using Sorted Array (2)

Inserting a new element ... eg: ***`insert(A00099999, "foo")`***

1	A00043522	Jimmy
2	A00066666	beelzebub
3	A00100000	186A0
4	A00111111	Bob
5	A00123456	n(n+1)/2
6	A00444444	bertcubed
7	A00666666	Beelzebub
8		
9		
10		

Using Sorted Array (3)

Inserting a new element ... eg: ***insert(A00099999, "foo")***

1	A00043522	Jimmy
2	A00066666	beelzebub
3	A00100000	186A0
4	A00111111	Bob
5	A00123456	n(n+1)/2
6	A00444444	bertcubed
7	A00666666	Beelzebub
8		
9		
10		

find location

- **(use binary search)**
- **O(logn) operation**

Using Sorted Array (4)

Inserting a new element … eg: ***insert(A00099999, "foo")***

1	A00043522	Jimmy
2	A00066666	beelzebub
3		
4	A00100000	186A0
5	A00111111	Bob
6	A00123456	n(n+1)/2
7	A00444444	bertcubed
8	A00666666	Beelzebub
9		
10		

find location

- **(use binary search)**
- **O(logn) operation**

create space

- **(move existing elements)**
- **O(n) operation**

Using Sorted Array (5)

Inserting a new element … eg: ***insert(A00099999, "foo")***

1	A00043522	Jimmy
2	A00066666	beelzebub
3	A00099999	foo
4	A00100000	186A0
5	A00111111	Bob
6	A00123456	n(n+1)/2
7	A00444444	bertcubed
8	A00666666	Beelzebub
9		
10		

find location
- **(use binary search)**
- **O(logn) operation**

create space
- **(move existing elements)**
- **O(n) operation**

put the new element
- direct access to array
- O(1) operation

Overall efficiency is:

O(logn)+O(n)+O(1) = O(n)

Using Sorted Array (6)

- *Search/retrieval* is $\boldsymbol{O(logn)}$
- *Insertion/deletion* is $\boldsymbol{O(n)}$

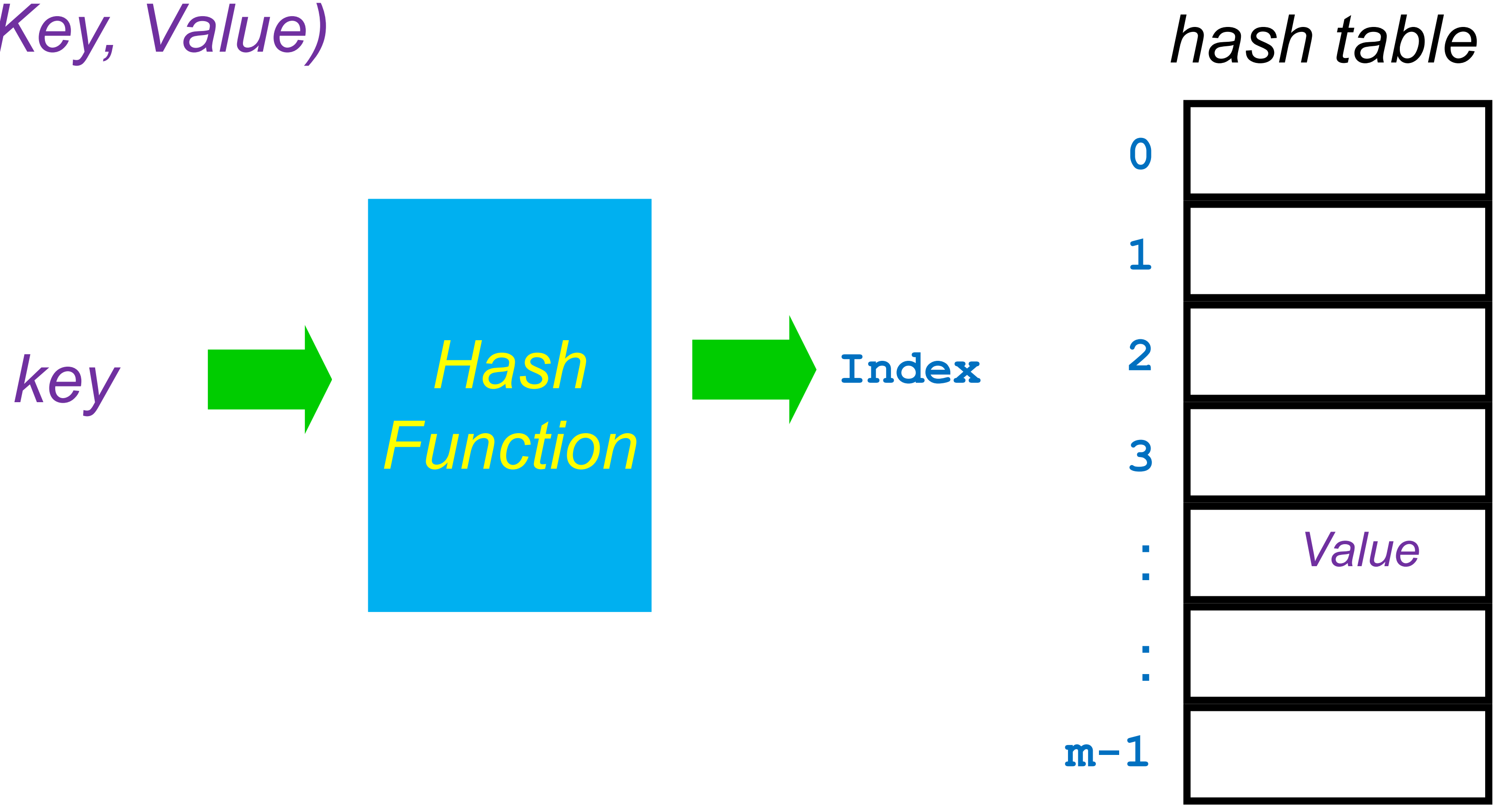
What if we use an unsorted Array:

- *Insertion* will be much faster – O(1)
- *Search, retrieve* will be slower – O(n)
- *Deletion* will be the same O(n)

- *So how to get better performance ... ?*
  - *Hashing*

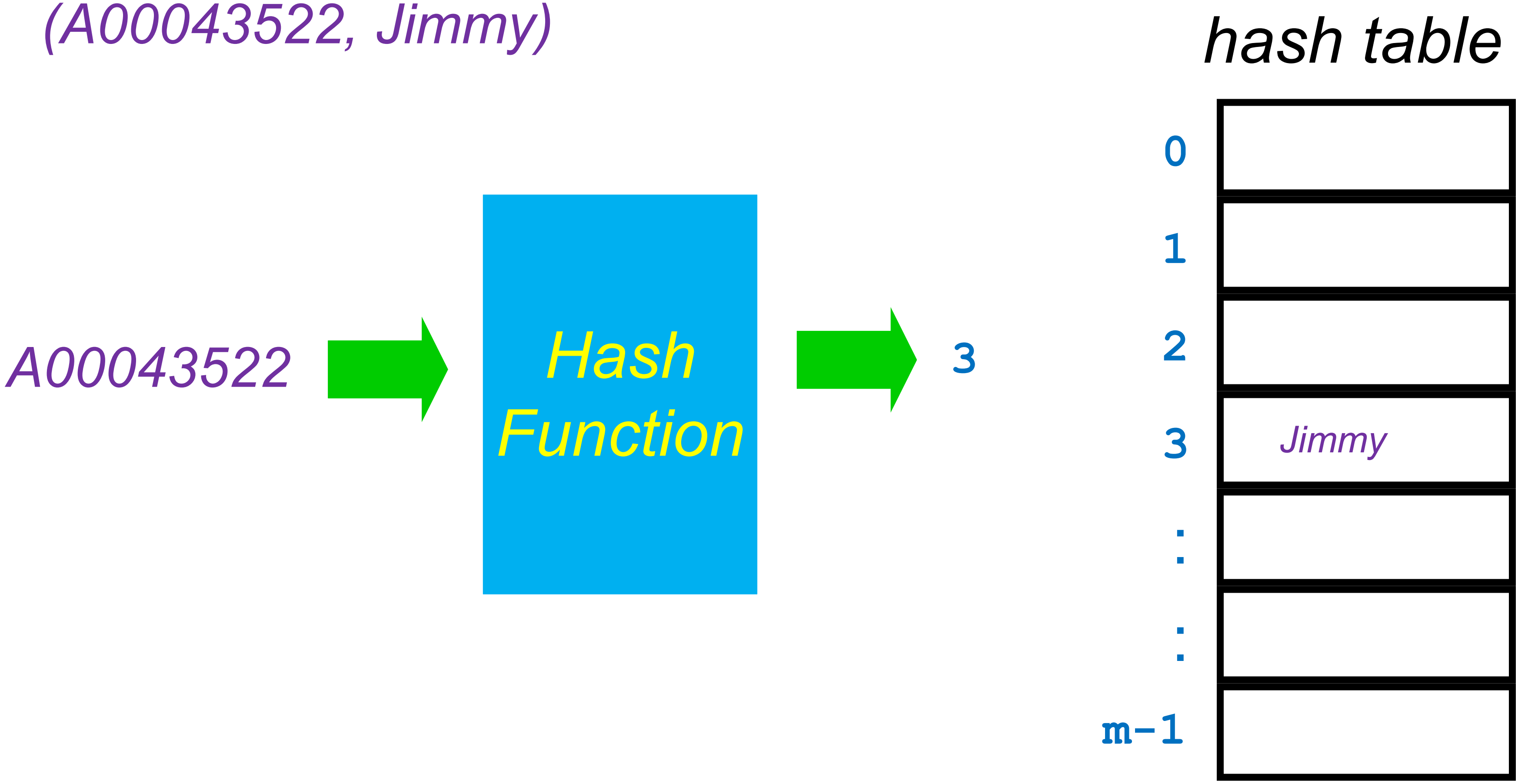
Hashing/ Hash Table

(Key, Value)



Example

(A00043522, Jimmy)



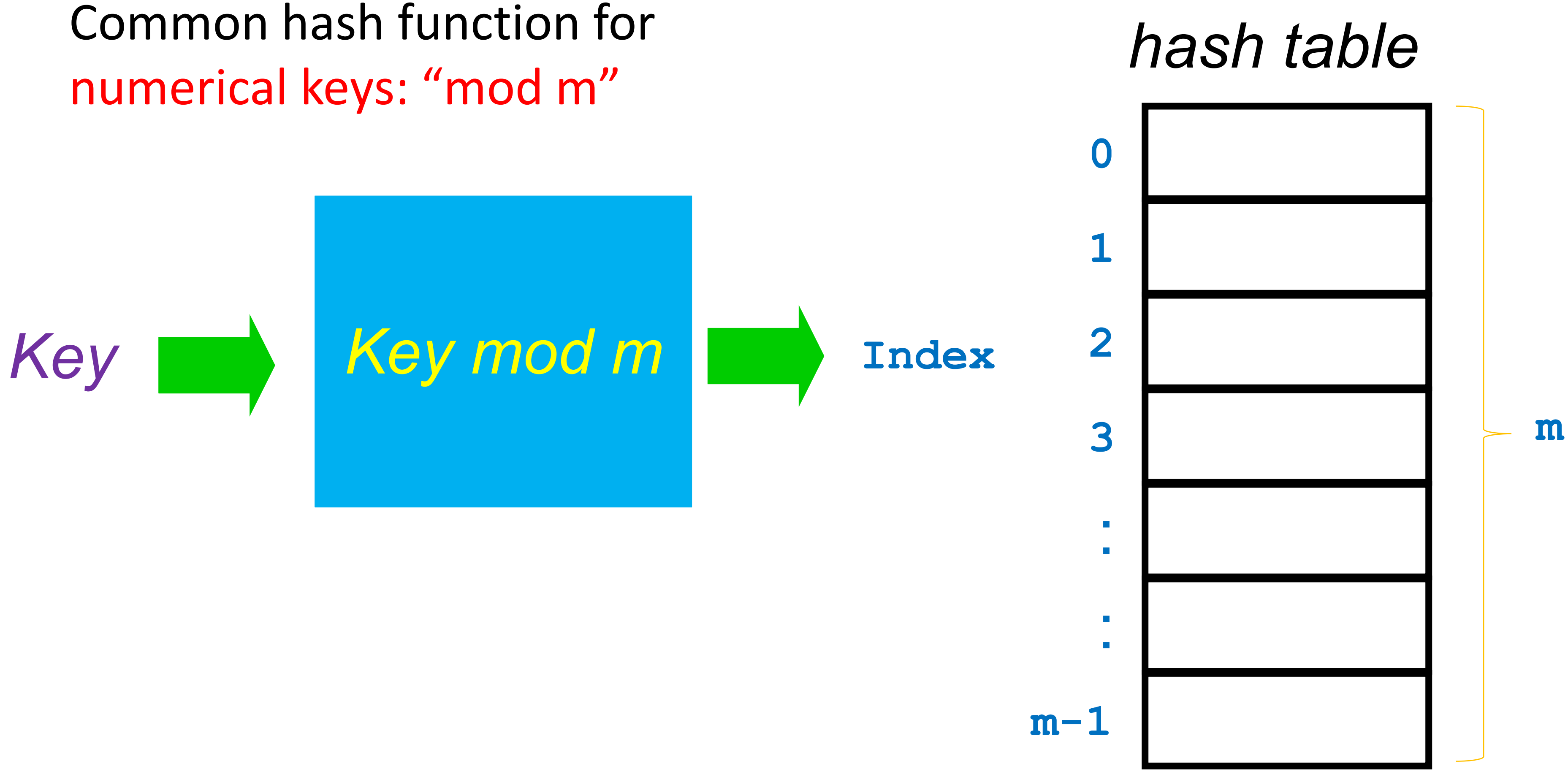
Hashing

- Each item has a unique key.
- Use a large array called a Hash Table.
- Use a Hash Function that maps keys to an index in the Hash Table.

$$f(key) = index$$

Hash Functions

Common hash function for
numerical keys: “mod m”

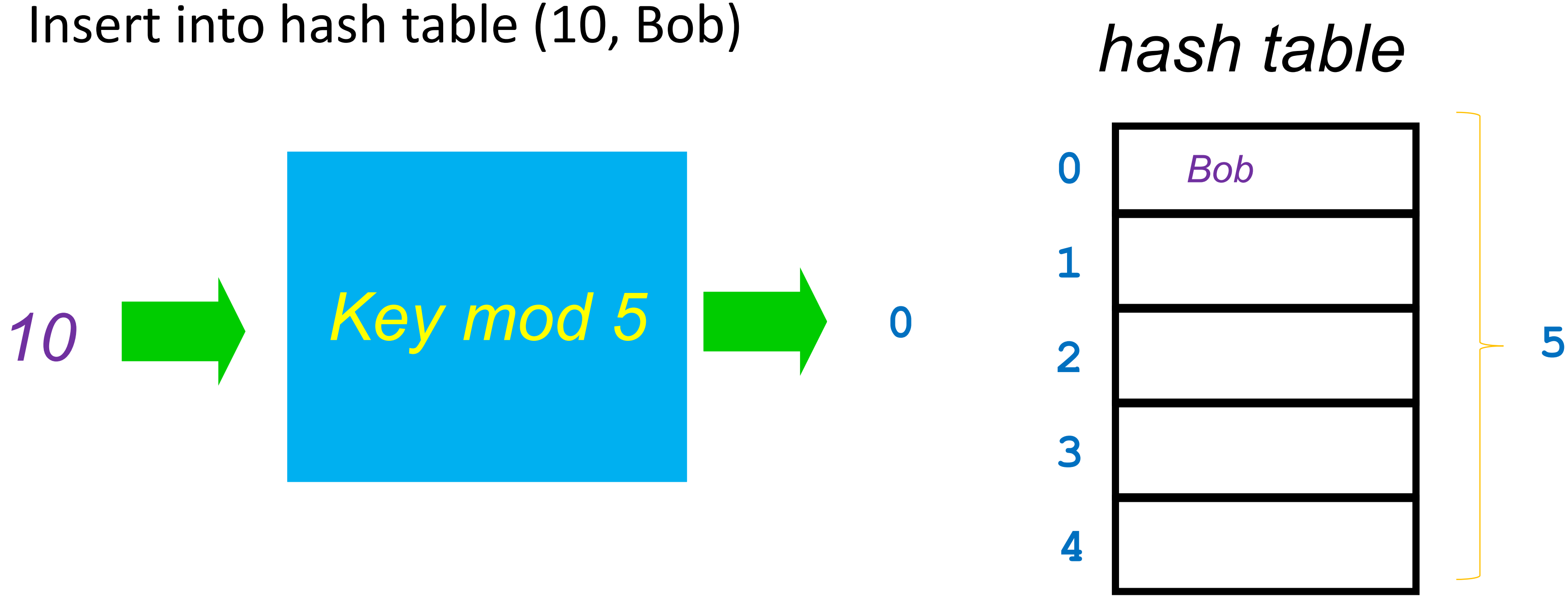


Hash Functions

Example

assume m=5

Insert into hash table (10, Bob)



Hash Functions

- What do we do if our key is not a number?
  - *answer: map it to a number!*

- Example

  assume m=5

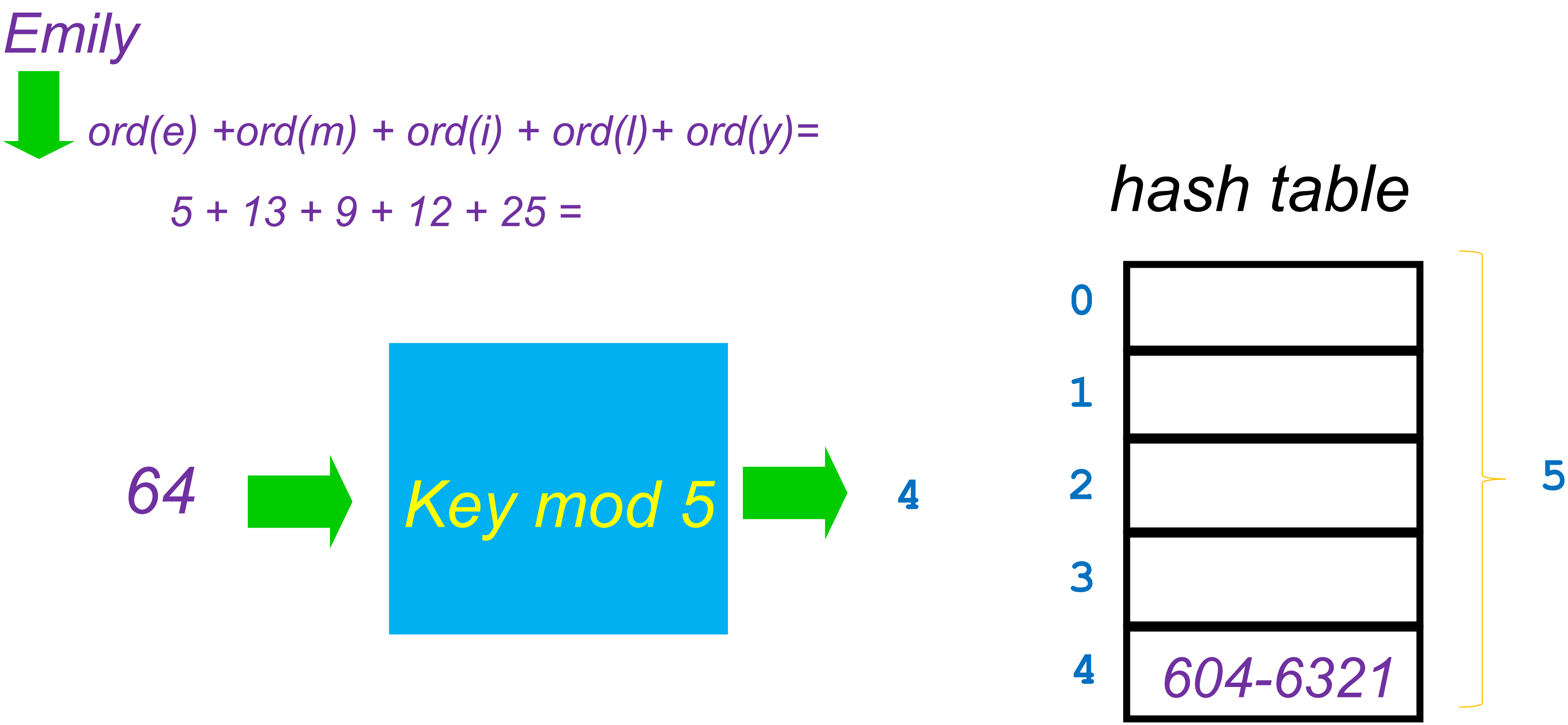
  Insert into hash table (Emily, 604-6321)

Hash Functions

Example

assume m=5

Insert into hash table (Emily, 604-6321)



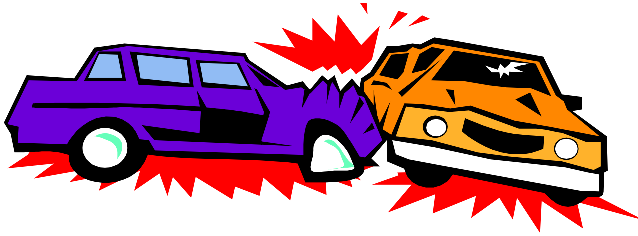
Hash Functions

- Sample hash function for string keys:

```
h ← 0                              // input is a string S of length s
for i ← 0 to s-1 do                // c_i is the char in i^th posn of S
    h ← h + ord(c_i)               // ord(c_i) is the relative posn
                                   //         of c_i in the alphabet
hashcode ← h mod numBuckets        // map sum of posns into range
```

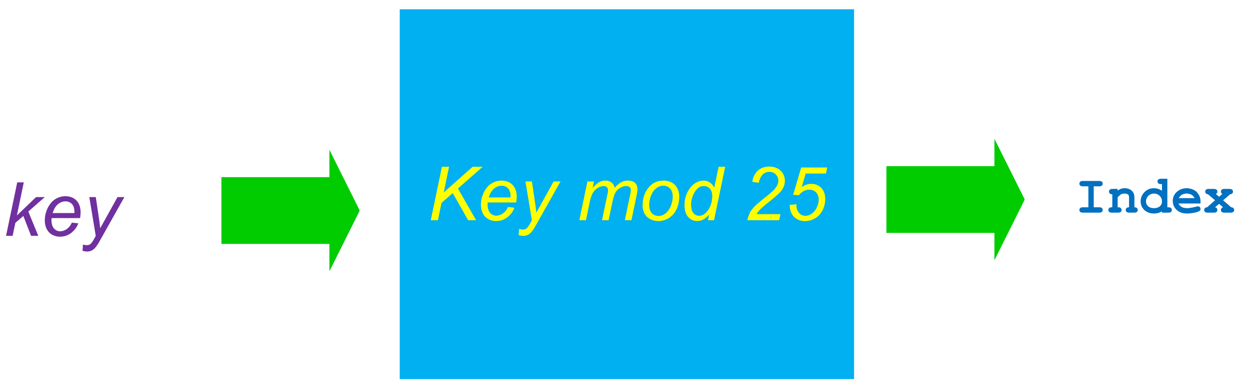
the actual hashcode depends on the number of buckets

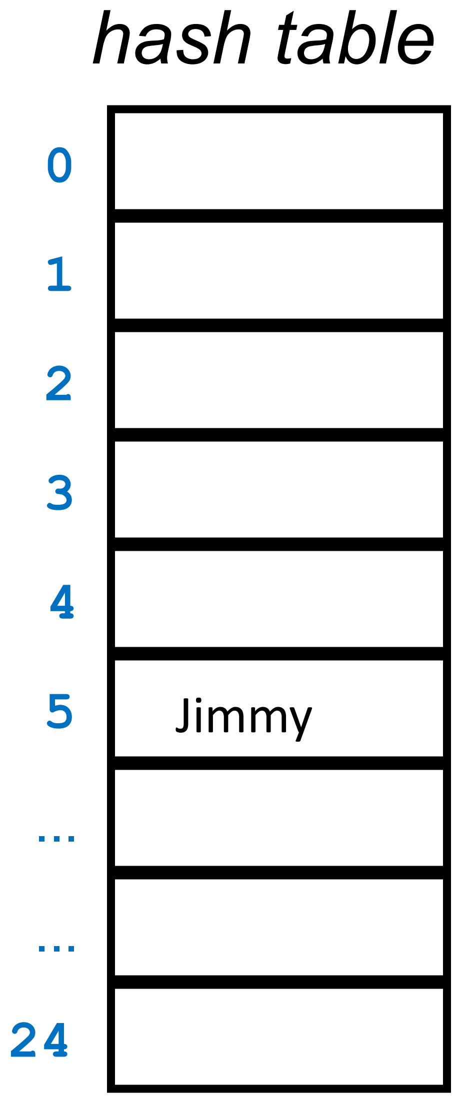
- This is similar to "H1" on the Lab



Collisions

Collisions occur when different keys are mapped to the same bucket





1.Insert into hash table (30, Jimmy)
index = 30 mod 25 = 5

2. Insert into hash table (105, Anthony)
index = 105 mod 25 = 5

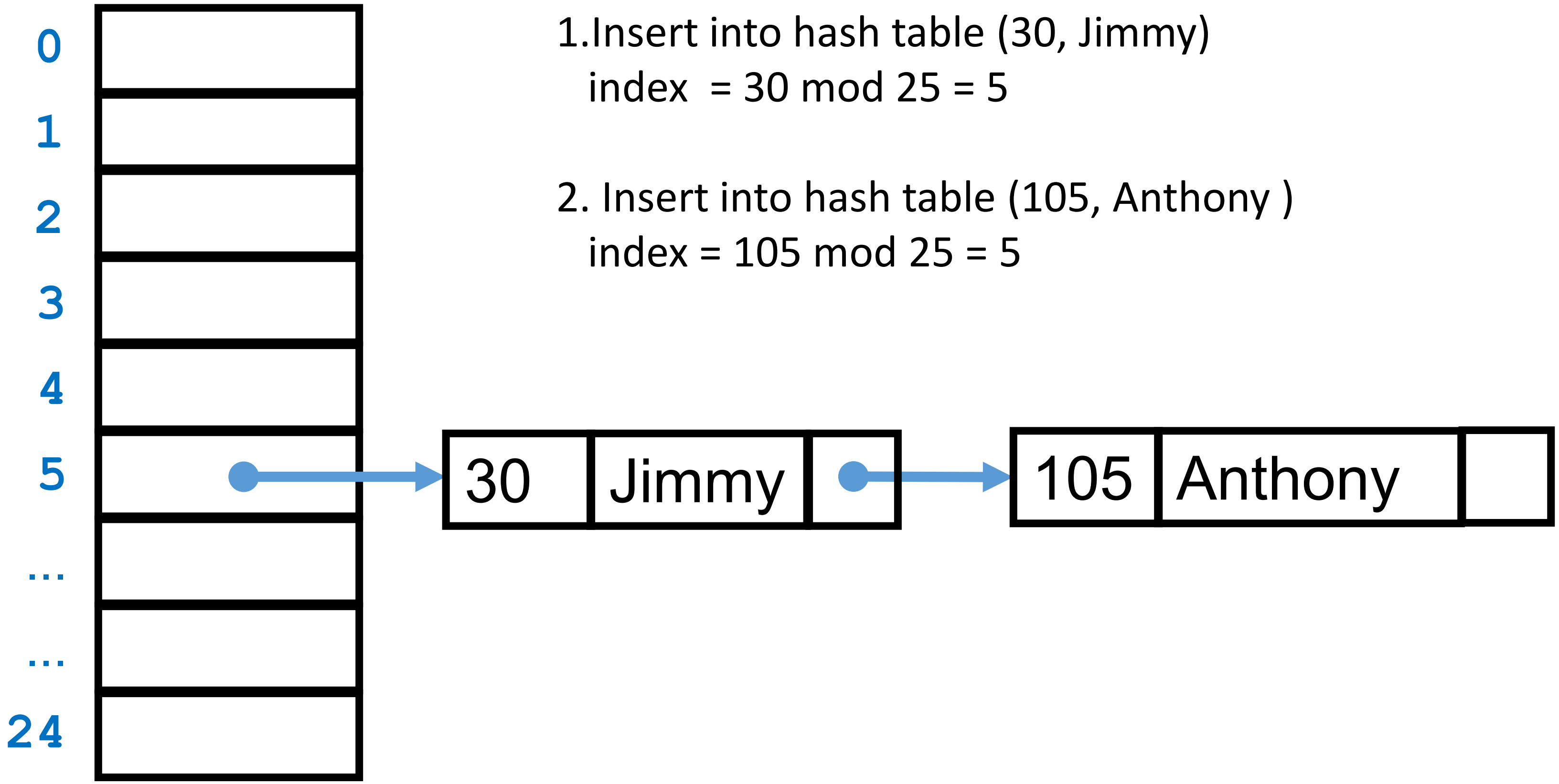
Collision Handling

Two strategies to handle collisions:

1. Separate Chaining
2. Closed Hashing

Collision Handling: Separate Chaining

- Each bucket in the table points to a *list* of entries that map there



1.Insert into hash table (30, Jimmy)
index = 30 mod 25 = 5

2. Insert into hash table (105, Anthony)
index = 105 mod 25 = 5

Separate chaining Example 1

- Use the hash function h(i) = i mod 7
- Draw the Separate chaining hash table resulting from inserting following keys and values:

  (44, red)

  (12, orange)

  (23, yellow)

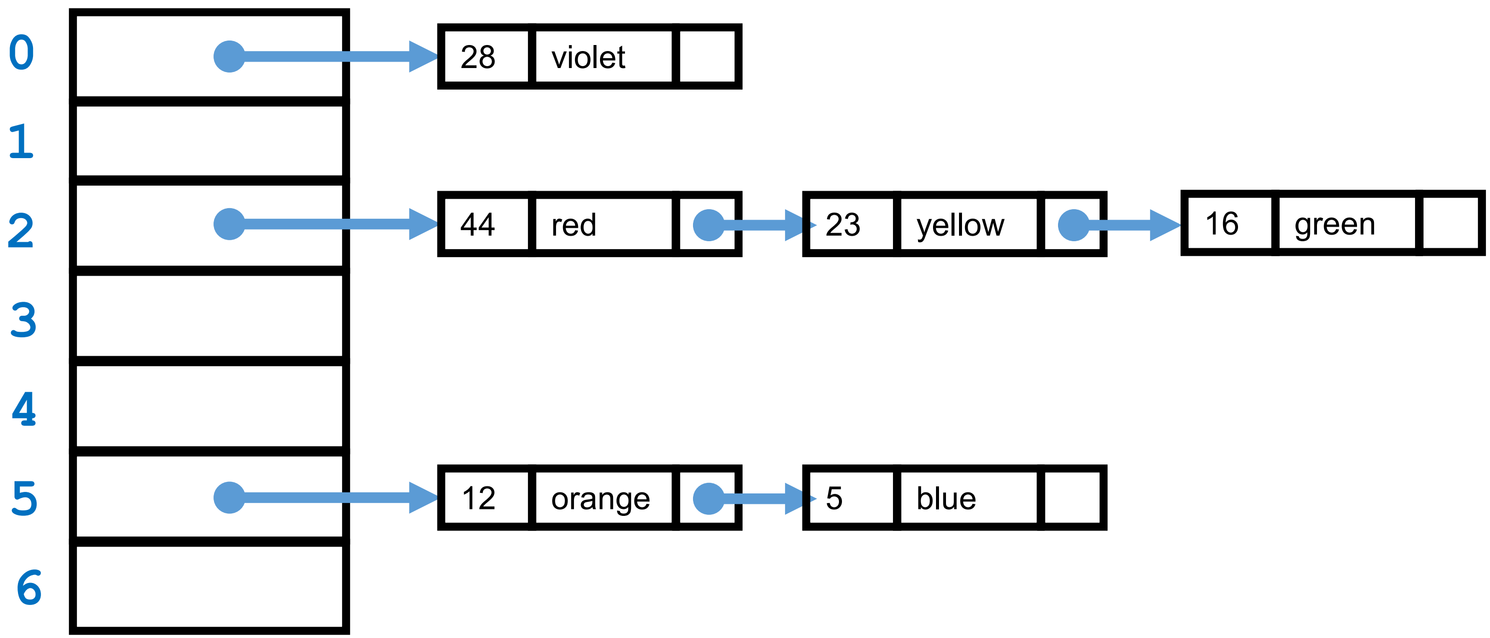
  (16, green)

  (5, blue)

  (28, violet)

(44, red)
(12, orange)
(23, yellow)
(16, green)
(5, blue)
(28, violet)

hash function h(i) = i mod 7

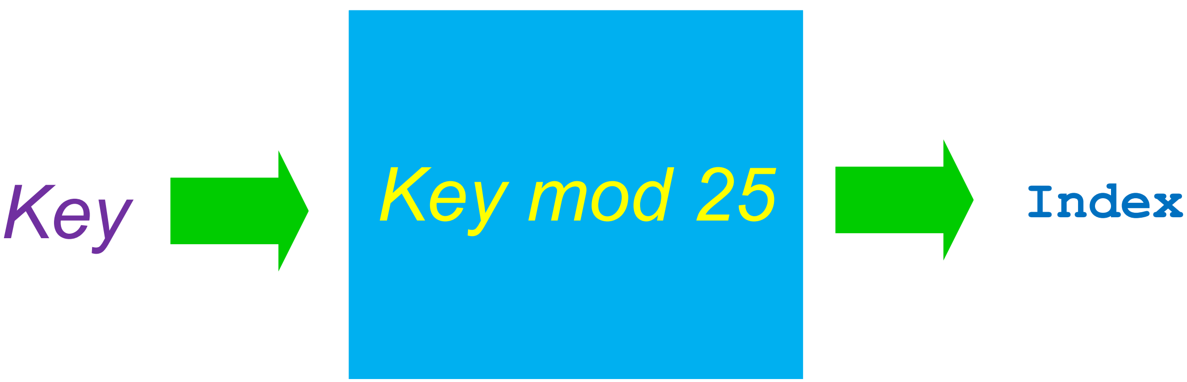


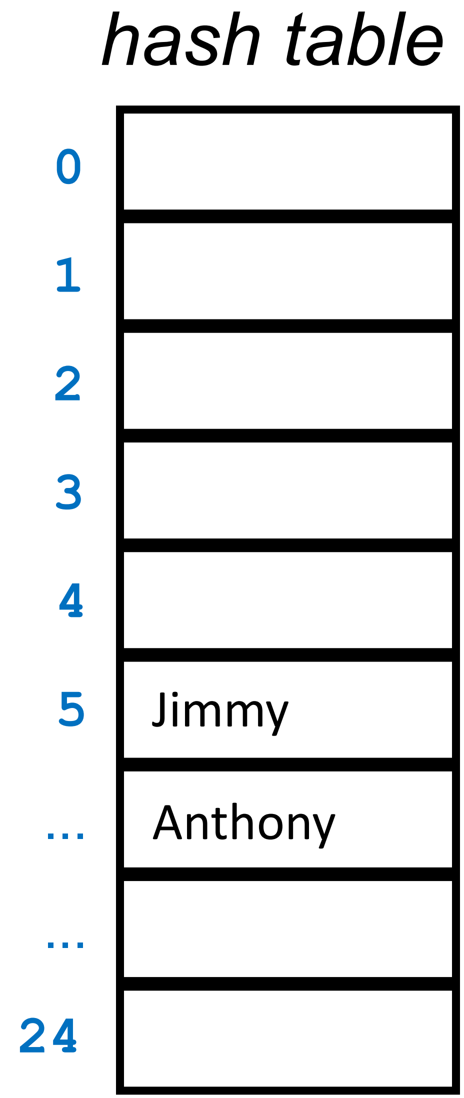
Strategy 2: Closed Hashing with Linear Probing

- It works like this:
  - compute the hash
  - if the bucket is empty, store the value in it
  - if there is a collision, linearly scan for next free bucket and put the key there
    - note: treat the table as a circular array

- Note: important - with this technique the size of the table must be at least n (or there would not be enough room!)

Linear Probing





1.Insert into hash table (30, Jimmy)
index = 30 mod 25 = 5

2. Insert into hash table (105, Anthony)
index = 105 mod 25 = 5

Closed Hashing Exercise

- Use the hash function h(i) = i mod 10
- Draw the hash table resulting from inserting following key and values:

  (44, mojo)

  (12, buzz)

  (13, iggy)

  (88, flem)

  (23, sue)

  (16, vern)

  (22, sami)

(44, mojo)
(12, buzz)
(13, iggy)
(88, flem)
(23, sue)
(16, vern)
(22, sami)

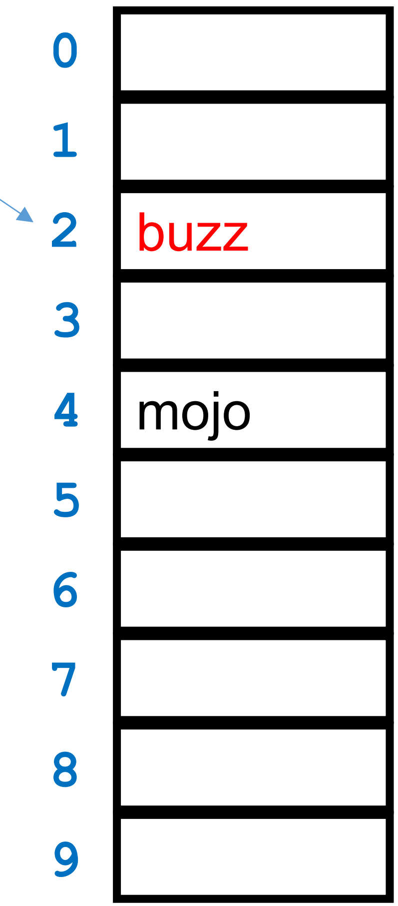
hash function h(i) = i mod 10

44 mod 10 = 4

(44, mojo)
(12, buzz)
(13, iggy)
(88, flem)
(23, sue)
(16, vern)
(22, sami)

hash function h(i) = i mod 10

12 mod 10 = 2



(44, mojo)
(12, buzz)
(13, iggy)
(88, flem)
(23, sue)
(16, vern)
(22, sami)

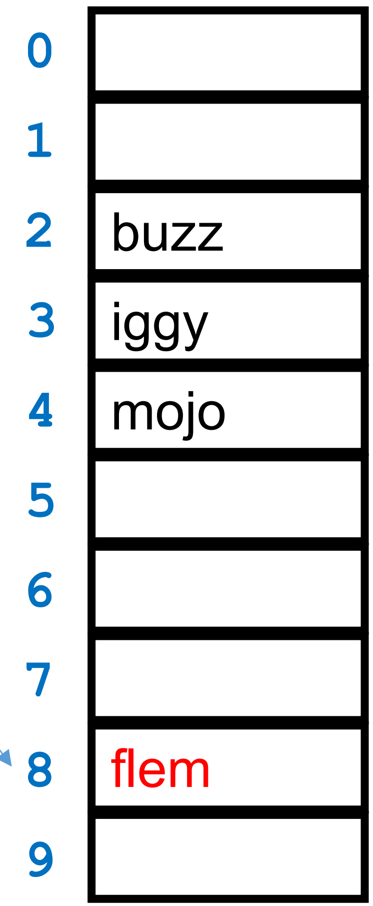
hash function h(i) = i mod 10

13 mod 10 = 3

(44, mojo)
(12, buzz)
(13, iggy)
(88, flem)
(23, sue)
(16, vern)
(22, sami)

hash function h(i) = i mod 10

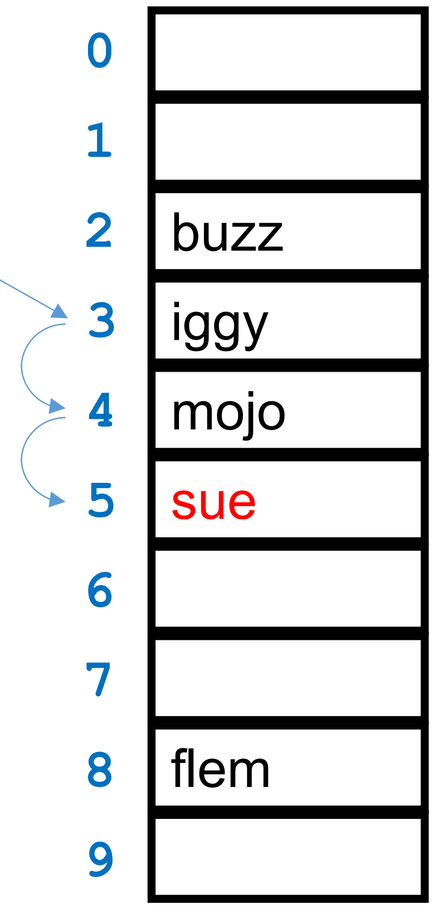
88 mod 10 = 8



(44, mojo)
(12, buzz)
(13, iggy)
(88, flem)
(23, sue)
(16, vern)
(22, sami)

hash function h(i) = i mod 10

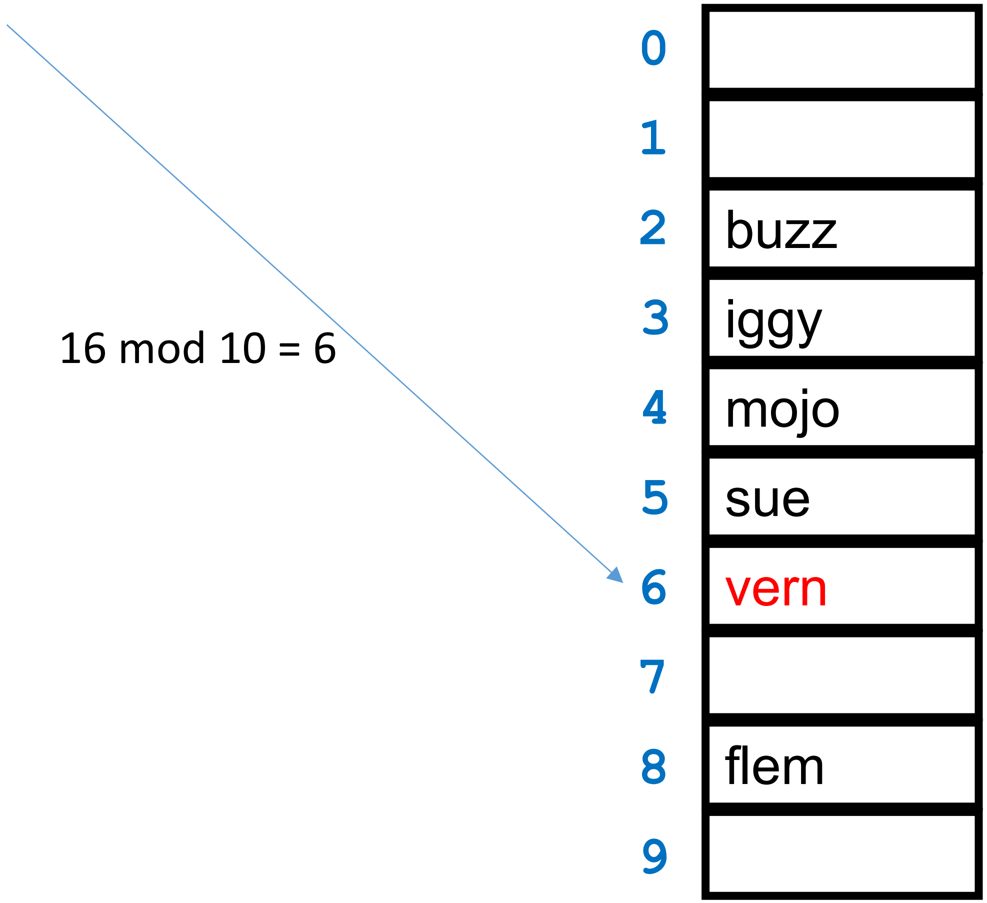
23 mod 10 = 3



Collision!
Followed by 2 probes

(44, mojo)
(12, buzz)
(13, iggy)
(88, flem)
(23, sue)
(16, vern)
(22, sami)

hash function h(i) = i mod 10

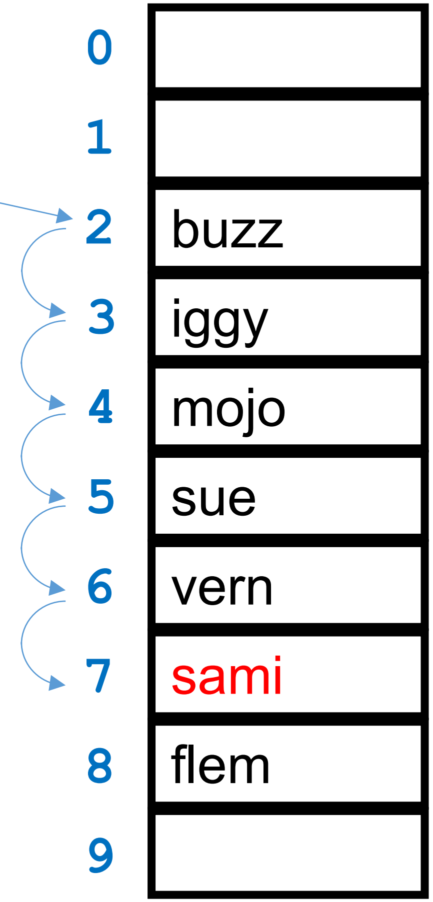


(44, mojo)
(12, buzz)
(13, iggy)
(88, flem)
(23, sue)
(16, vern)
(22, sami)

hash function h(i) = i mod 10

22 mod 10 = 2

Collision!
Followed by 5 probes



Efficiency of Hashing

What is the efficiency of the hashtable structure?

- **insert**(key, value) ... is (usually) $\boldsymbol{O(1)}$
- value $\leftarrow$ **get**(key) ... is (usually) $\boldsymbol{O(1)}$
- **delete**(key) ... is (usually) $\boldsymbol{O(1)}$

- Warning: there *could* always be a degenerate case, where (almost) every insert causes a collision to be handled. We could end up with O(n) for each operation.

$\rightarrow$***conclusion : implementation of the hashing function is important***

$\rightarrow$***it must distribute the keys evenly over the buckets***

Hash Functions

- the efficiency of hashing depends on the quality of the **hash function**

  A “good” hash function will

  1. distribute the keys uniformly over the buckets
  2. produce very different hashcodes for similar data

- hashing of numbers is relatively easy, as we just distribute them over the buckets with

  key mod *numBuckets*

Hashing Strings

- most keys are Strings, and Strings are a bit trickier
  - consider the simple algorithm:

```
h ← 0
for i ← 0 to s-1 do
    h ← h + ord(c_i)    // ord(c_i) is the posn of char i
code ← h mod numBuckets
```

- Is that a good hash function?
  - sample: assume numBuckets = 99
    - hash("dog") = 26
    - hash("god") = 26
    - hash("add") = 9
    - hash("dad") = 9

Better String Hash Function

- a better hashcode algorithm for strings

```
alpha  ← |alphabet|       // size of the alphabet used
h ← 0
for i ← 0 to s-1 do
    h ← h + (ord(c_i) * alpha^(i))
code ← h mod numBuckets
```

  - Assuming alpha = 128 (number of ascii codes)
  - Assuming numBuckets = 99
    - dog = 89
    - god = 44
    - add = 16
    - dad = 31

- This is similar to our “H2” on the Lab

Practice problems

1. Chapter 7.1, page 257, questions 3, 7
2. Chapter 7.2, page 267, question 1,2
3. Chapter 7.3, page 275, question 1,2,7