Space and Time Trade-Offs

(Chapter 7)

Space-time tradeoff

- **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.
- Best algorithm to solve a problem is one that
  - Requires less memory and
  - Takes less time to complete

In practice it is not always possible

Space-time tradeoff

- 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: Bubblesort)

- 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)

Space-for-time tradeoffs varieties

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

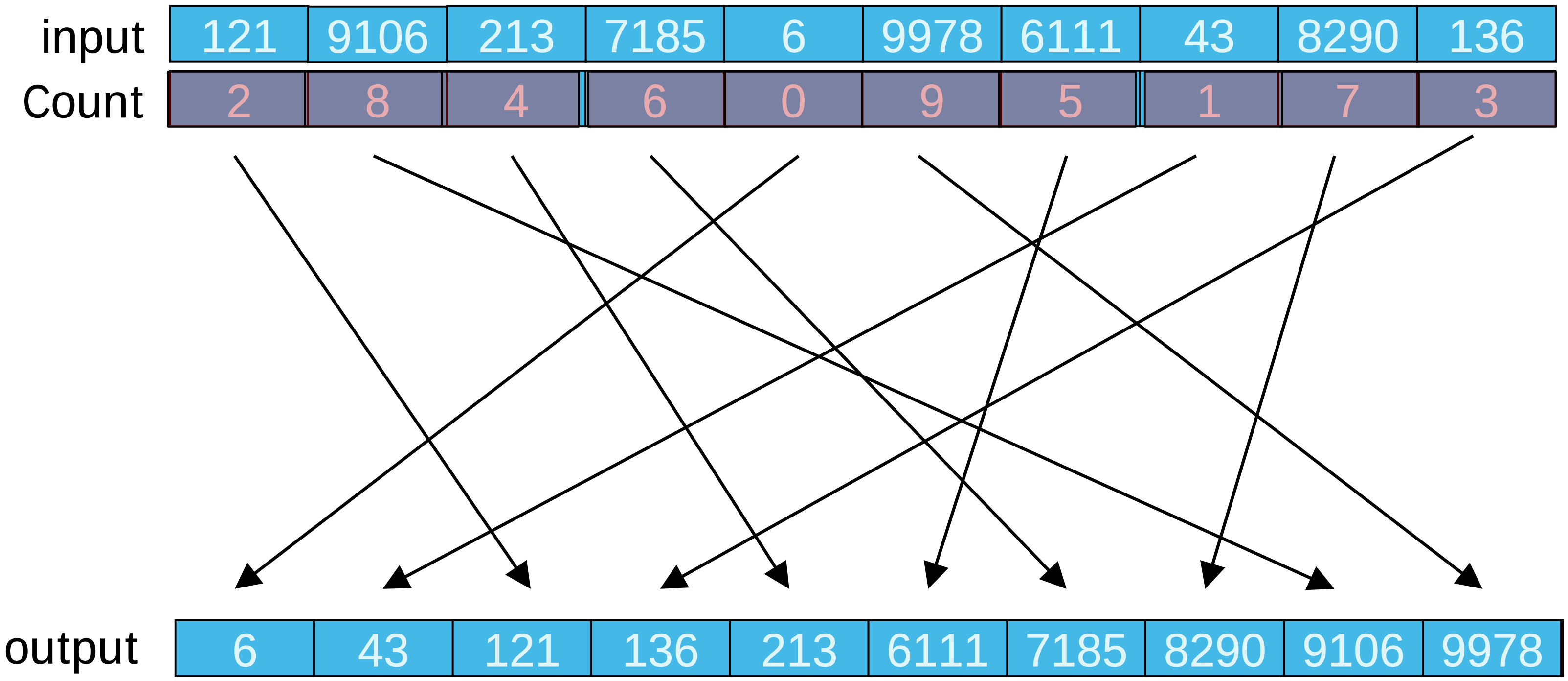
Comparison Counting Sort

- 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

- Move each input element to it's corresponding position



Comparison Counting Sort

```
Algorithm ComparisionCountingSort 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]
```

Comparison Counting Sort

- Efficiency:
  - it is $O(n^2)$
    - But of course we have other sorts (mergesort, heapsort) that are $O(n\log n)$

Space-for-time tradeoffs varieties

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

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

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

$C[j] \leftarrow 0$

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

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

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

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

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

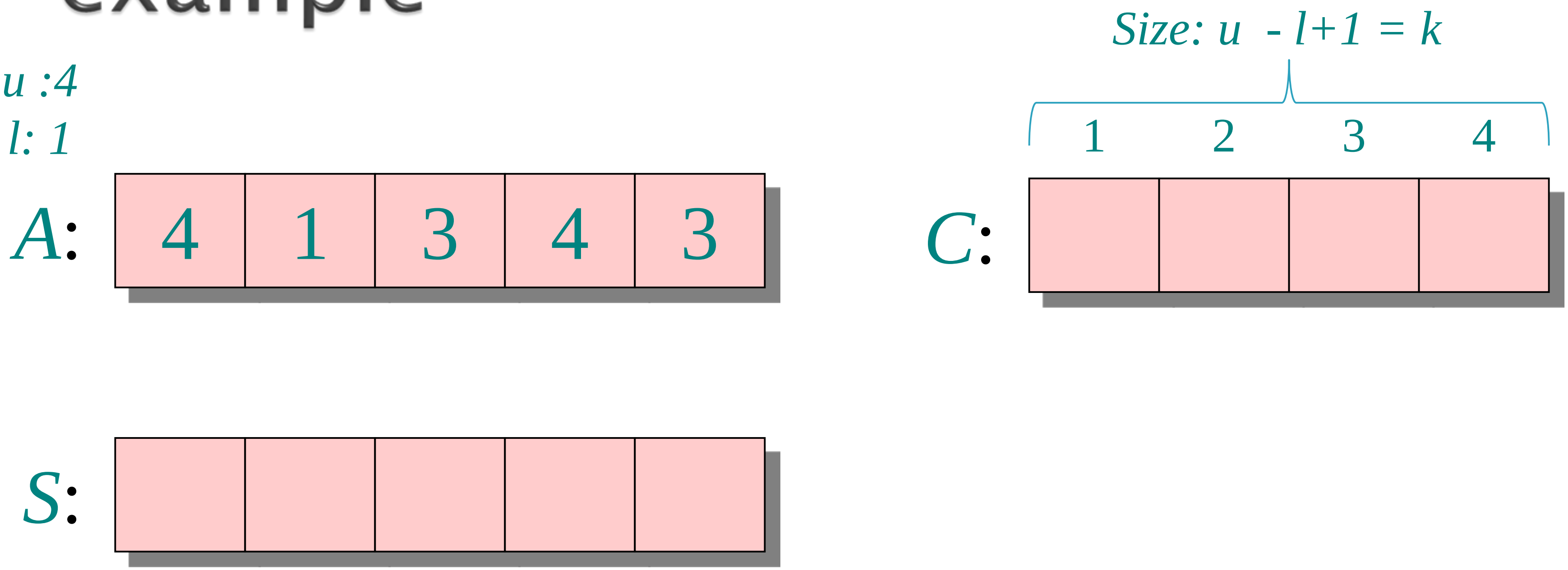
$j \leftarrow \mathrm{A}[\,i]\text{-}\,l$

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

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

return S

Distribution Counting Sort– example



Loop 1: initialization

A:

4	1	3	4	3

C:

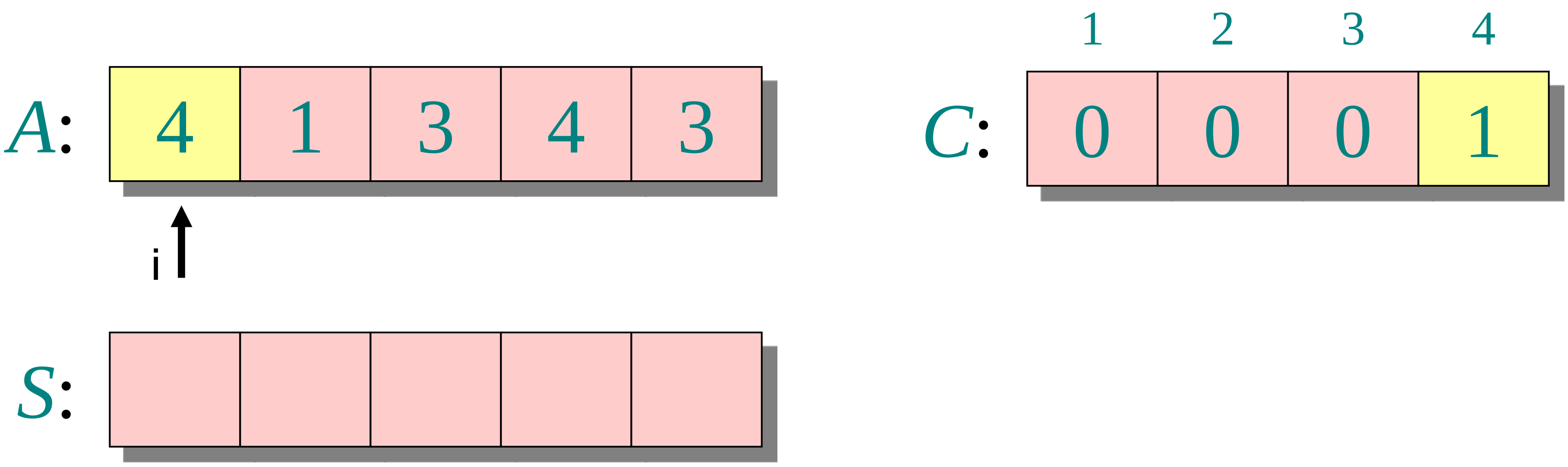
1	2	3	4
0	0	0	0

S:

1. **for** $j \leftarrow 0$ **to** u-l

  do $C[j] \leftarrow 0$

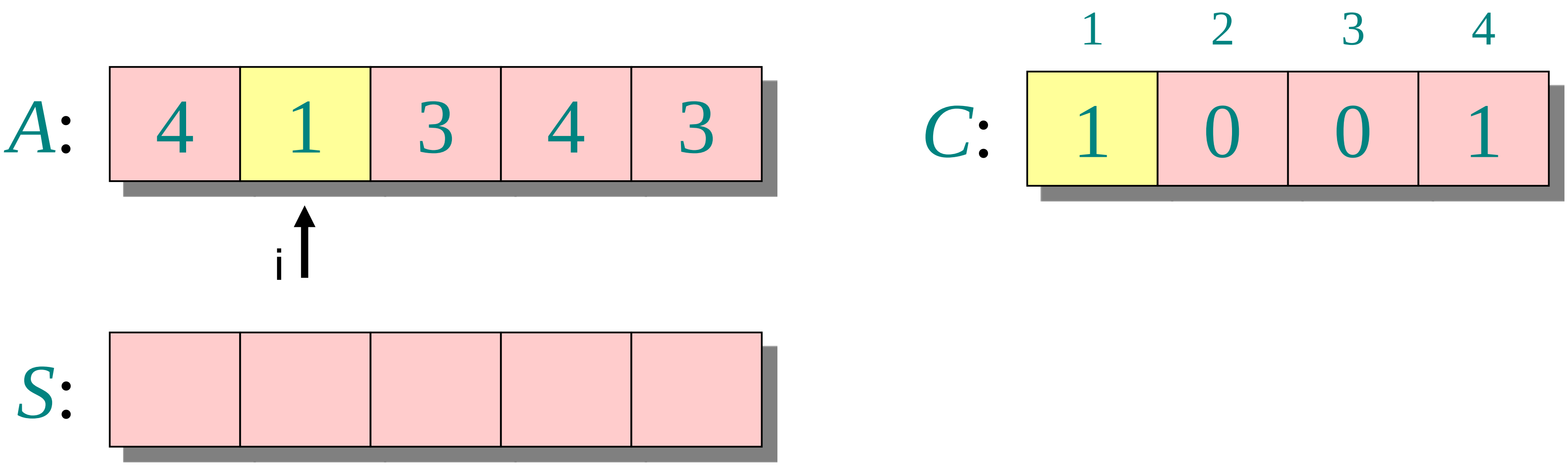
Loop 2: count



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

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

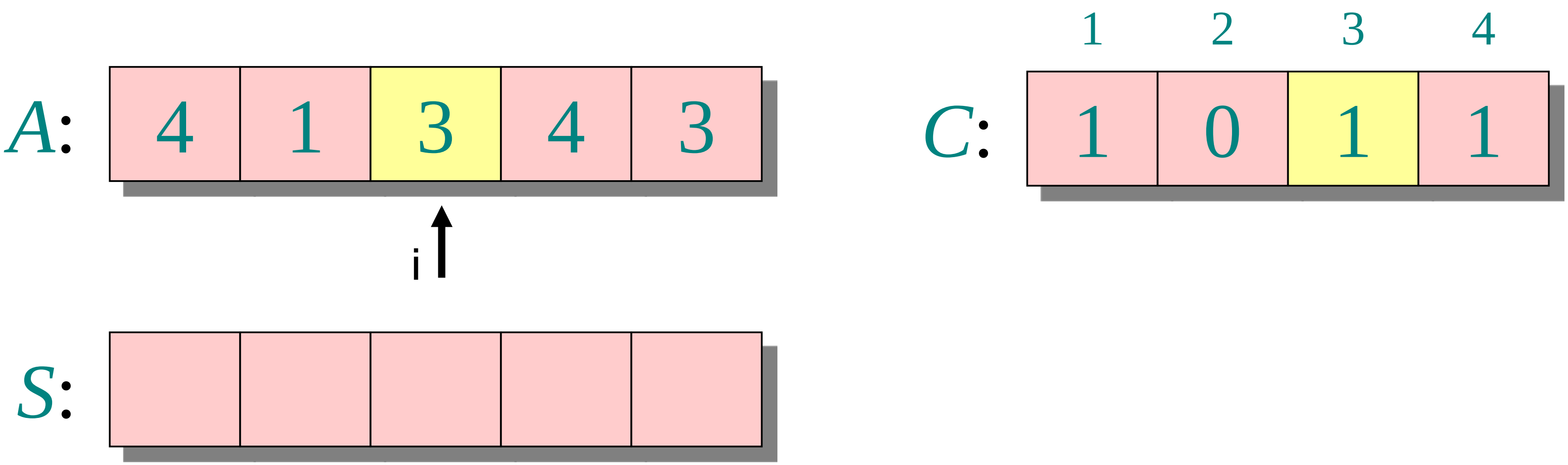
Loop 2: count



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

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

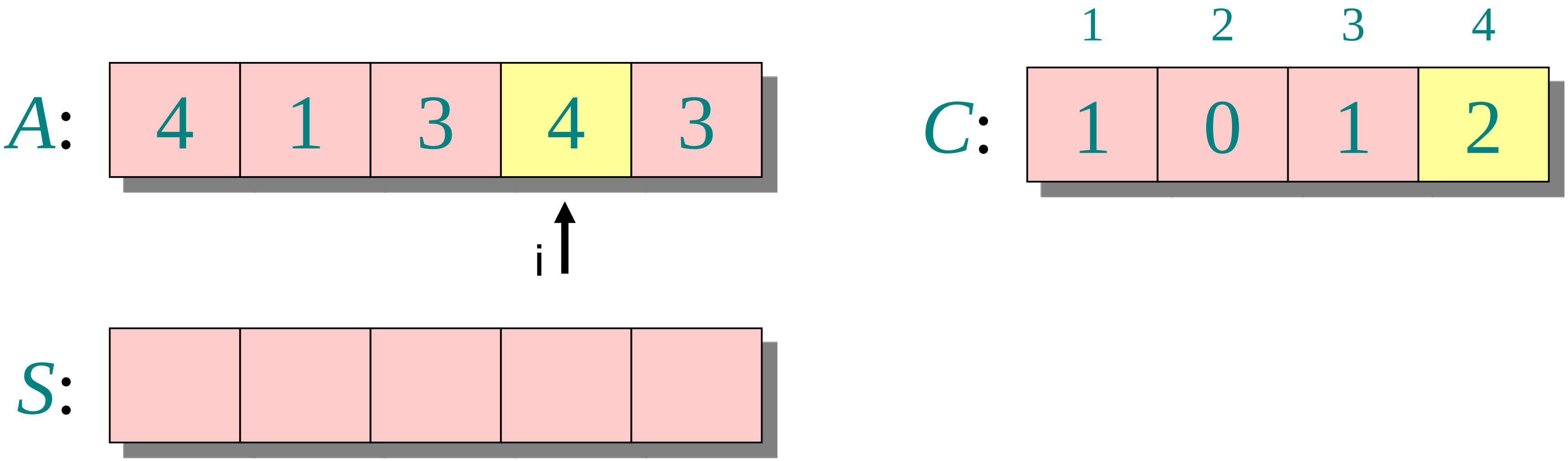
Loop 2: count



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

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

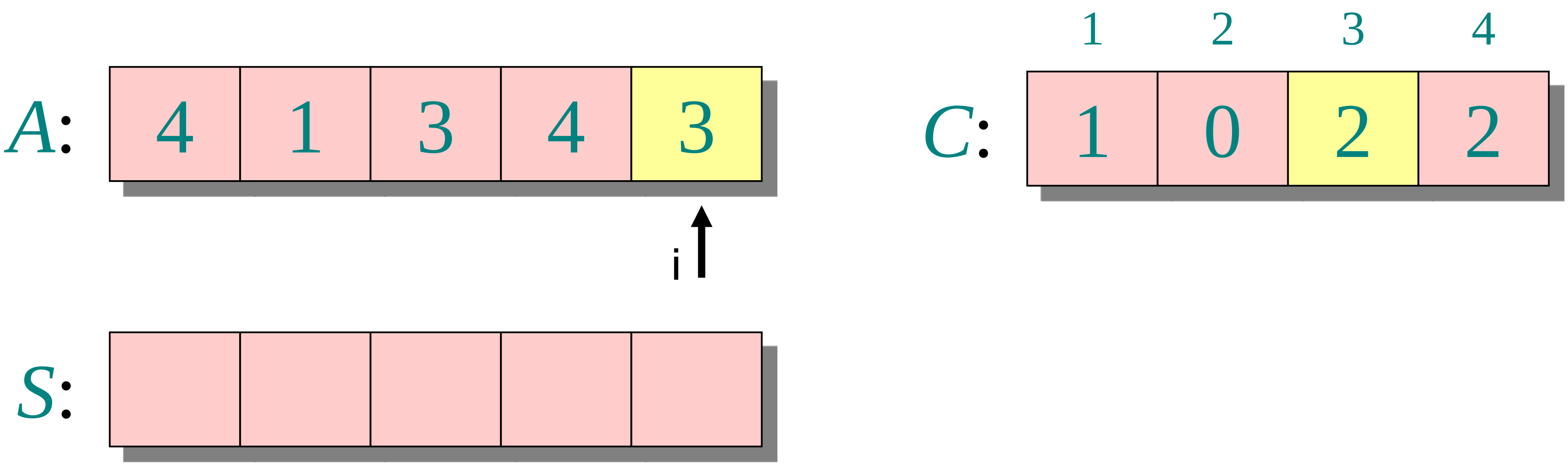
Loop 2: count



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

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

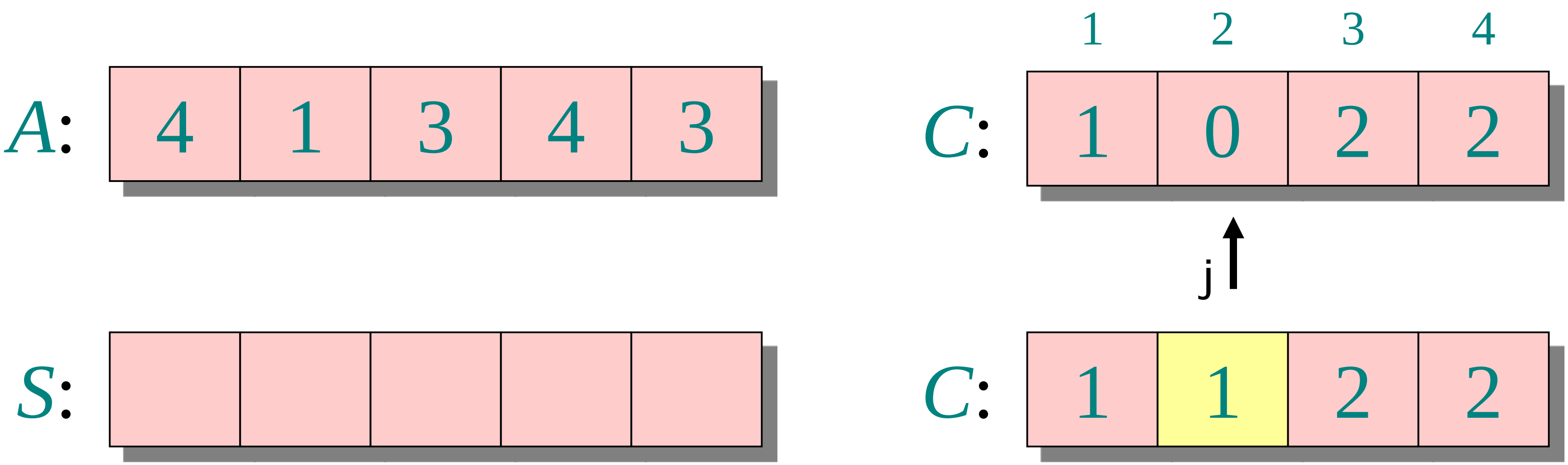
Loop 2: count



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

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

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 3: compute running sum

A:

4	1	3	4	3

C:

1	2	3	4
1	1	2	2
		j	

S:

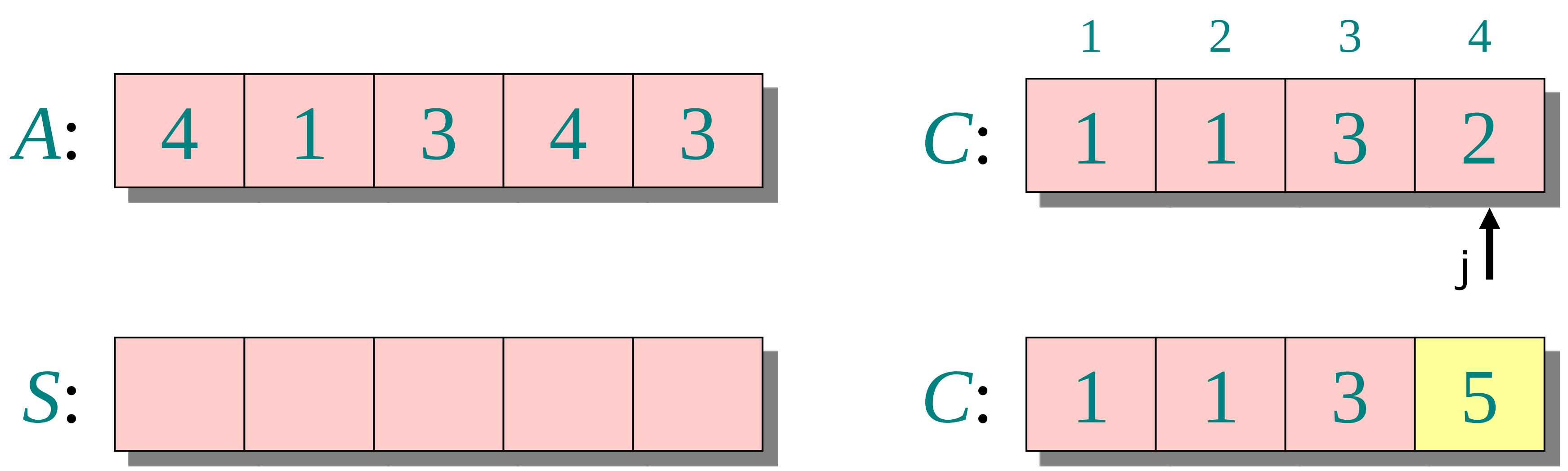
C:

1	1	3	2

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

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

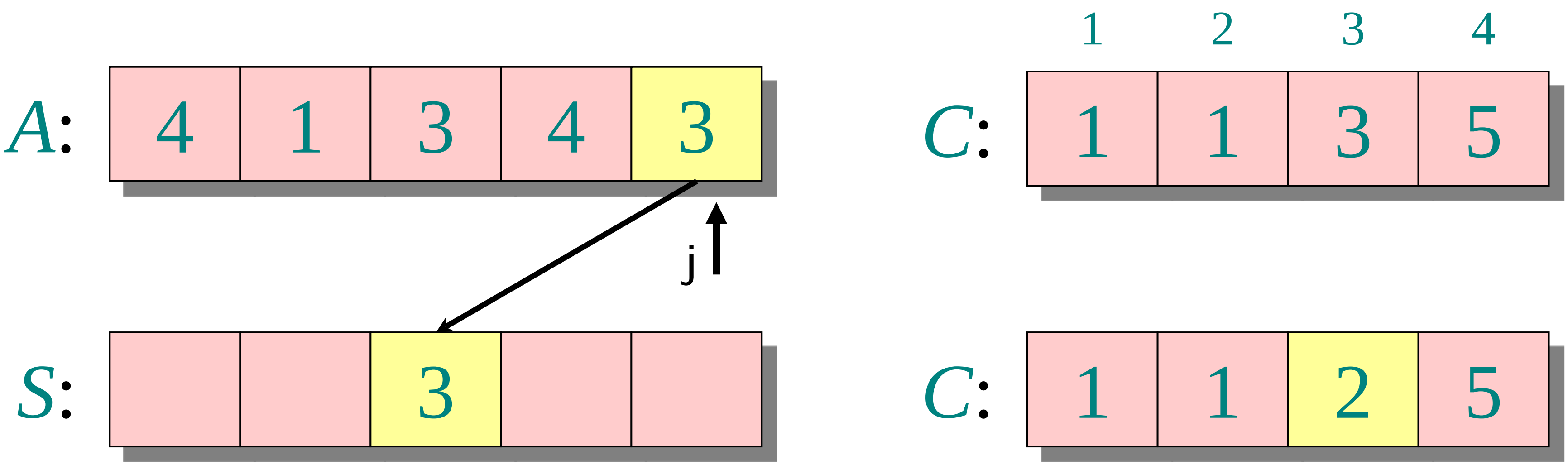
Loop 3: compute running sum



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

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

Loop 4: re-arrange



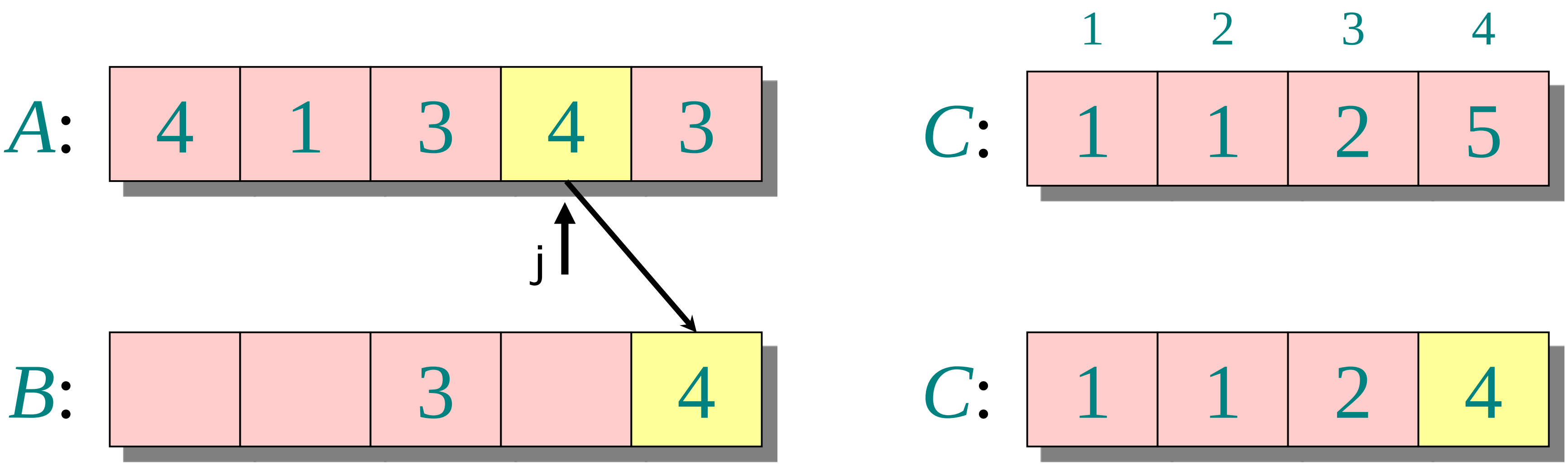
4. **for** $i \leftarrow n\text{-}1$ **downto** 0

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



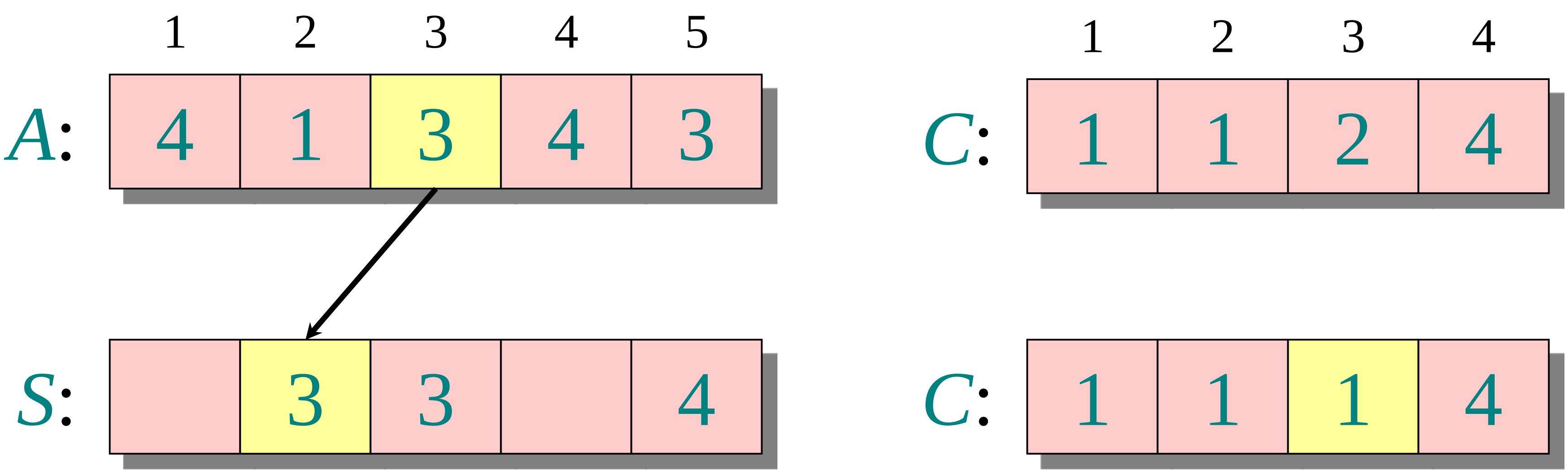
4. **for** $i \leftarrow n\text{-}1$ **downto** 0

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



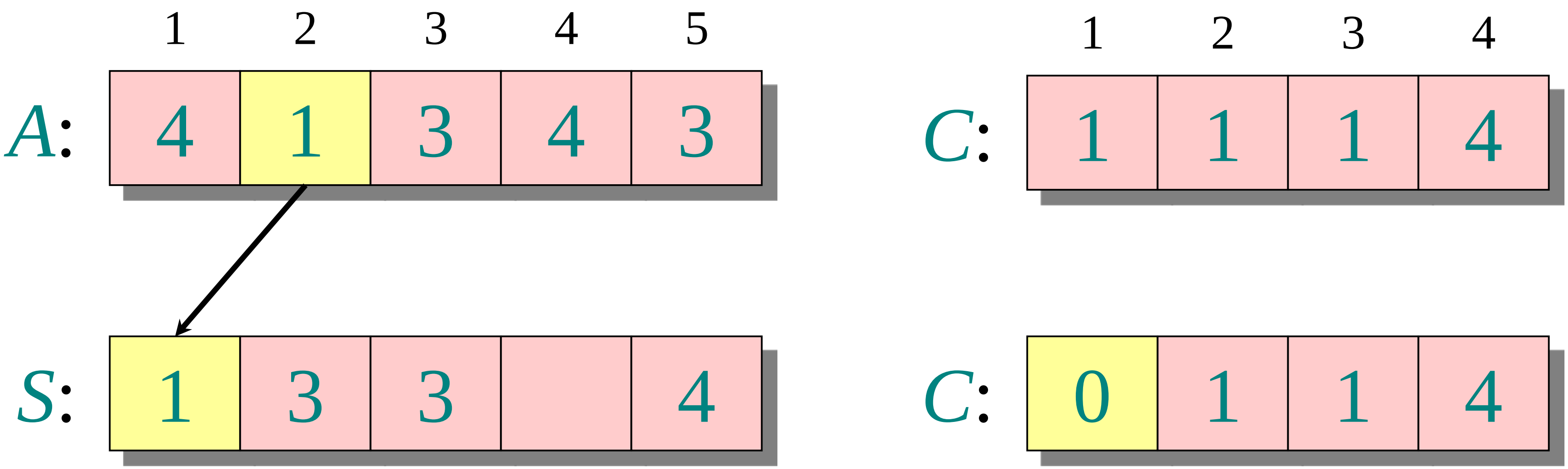
4. for $i \leftarrow n\text{-}1$ **downto** 0

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

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

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

Loop 4: re-arrange



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

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

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

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

Loop 4: re-arrange

	1	2	3	4	5
A:	4	1	3	4	3
S	1	3	3	4	4

	1	2	3	4
C:	0	1	1	4
C:	0	1	1	3

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

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

$S[D[j]-1] \leftarrow A[i]$

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

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

$O(k)$ $\left\{ \begin{array}{l} \textbf{for } j \leftarrow 0 \textbf{ to } u\text{-}l \textbf{ do} \\ \qquad C[j] \leftarrow 0 \end{array} \right.$

$O(n)$ $\left\{ \begin{array}{l} \textbf{for } i \leftarrow 0 \textbf{ to } n\text{-}1 \textbf{ do} \\ \qquad C[A[\,i]\text{-}l] \leftarrow C[A[\,i]\text{-}l] + 1 \end{array} \right.$

$O(k)$ $\left\{ \begin{array}{l} \textbf{for } j \leftarrow 1 \textbf{ to } u\text{-}l \textbf{ do} \\ \qquad C[j] \leftarrow C[j\text{-}1] + C[j] \end{array} \right.$

$O(n)$ $\left\{ \begin{array}{l} \textbf{for } i \leftarrow n\text{-}1 \textbf{ downto } 0 \textbf{ do} \\ \qquad j \leftarrow \mathrm{A}[\,i]\text{-}\,l \\ \qquad S[C[\,j]\,\text{-}1] \leftarrow \mathrm{A}[\,i] \\ \qquad C[\,j] \leftarrow C[\,j] - 1 \end{array} \right.$

$O(n + k)$ **return S**

Distribution Counting Sort

Analysis

- As long as the *range of valid input values is roughly less than or equal to the number of input values (n)*, **the algorithm is O(n)**

this is very good efficiency, better than mergesort

Space-for-time tradeoffs varieties

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

2. <u>**Pre-structuring:**</u> 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 string of *n* characters to search in

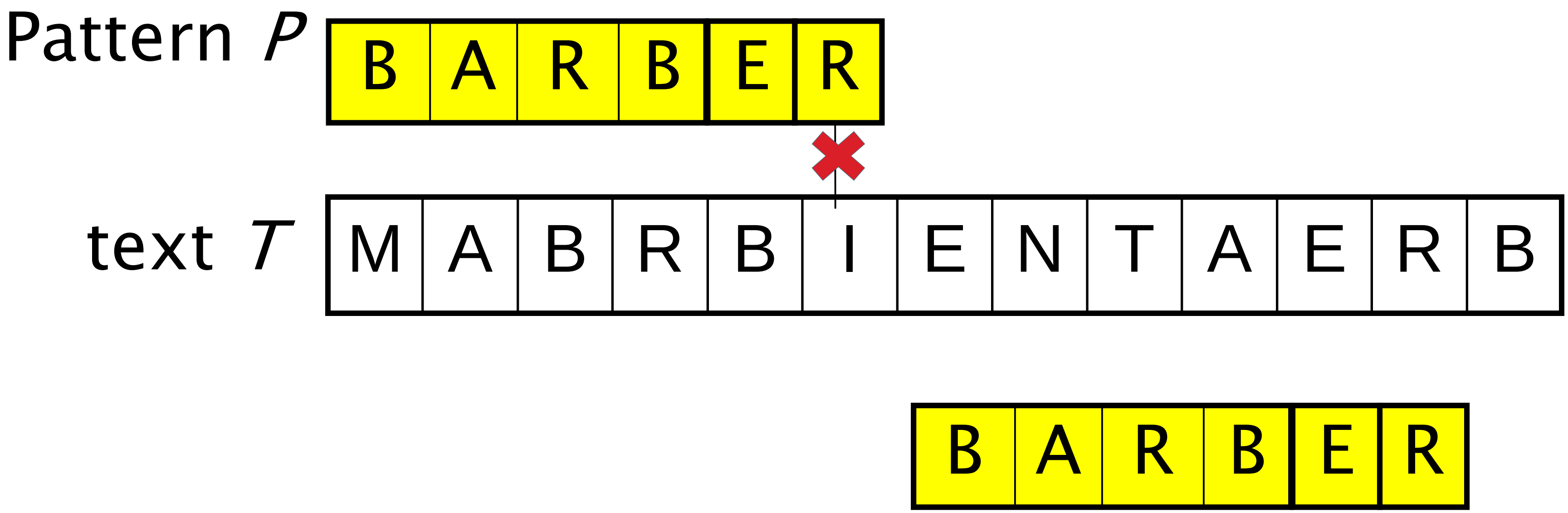
- **Brute force algorithm:**
  1. Align pattern at beginning of text
  2. Moving from left to right, compare each character of pattern to the corresponding character in text until
     - All characters are found to match (successful search); or
     - A mismatch is detected
  3. While pattern is not found and the text is not yet exhausted, realign 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?

- **key observation:** each time we have a "mismatch" (ie: a pattern char doesn't match the corresponding text char), we *may be able to shift more than one character* before starting to compare again

Input Enhancement in String Matching



- *Comparing the chars from* ***right to left***
- *There is no "I" in BARBER, so we should shift the pattern all the way past the "I"*
- Determines the umber 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:

$$s_0 \quad \dots \quad \quad \quad \quad c \quad \dots \quad s_{n-1}$$

B A R B E R

Starting with the last R of the pattern and moving right to left if a mismatch occurs shift to right by looking at character c

String Matching: Input Enhancement Cases

- Case1: If there are no *c*'s in the pattern

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

$\nparallel$

B A R B E R

B A R B E R

shift the pattern by its entire length

String Matching: Input Enhancement Cases

- Case2: If there are occurrences of character c in the pattern but it is not the last one there

$$
\begin{array}{cccccccccccc}
s_0 & \ldots & & & & & B & & & \ldots & s_{n-1} \\
 & & & & & & \not\parallel & & & & \\
 & & B & A & R & B & E & R & & & \\
 & & & & B & A & R & B & E & R &
\end{array}
$$

shift to align the rightmost occurrence of c in the pattern with the c in the text

String Matching: Input Enhancement Cases

- Case3: If *c* is the last char in the pattern, and occurs only once in the pattern

$$
\begin{array}{cccccccccccccc}
s_0 & \ldots & & & M & E & R & & & & & & \ldots & s_{n-1} \\
 & & & & \not\parallel & \parallel & \parallel & & & & & & & \\
 & & L & E & A & D & E & R & & & & & & \\
\end{array}
$$

$$
\begin{array}{cccccccc}
 & & & & & & L\ E\ A\ D\ E\ R
\end{array}
$$

shift the pattern by its entire length

String Matching: Input Enhancement Cases

- Case4: if *c* the last char in the pattern, and occurs multiple times in the pattern

$$
\begin{array}{cccccccccccccc}
s_0 & \dots & & & & & A & R & & & & \dots & s_{n-1} \\
 & & & & & & \nparallel & \parallel & & & & & \\
 & & R & E & O & R & D & E & R & & & & \\
 & & & & & R & E & O & R & D & E & R &
\end{array}
$$

shift to align the rightmost occurrence of c in the pattern with the c in the text

The Strategy

- How can we use this observation for input enhancement?

- Strategy:
  - we are going to create a “shift table”.
    - It will have 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!)

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

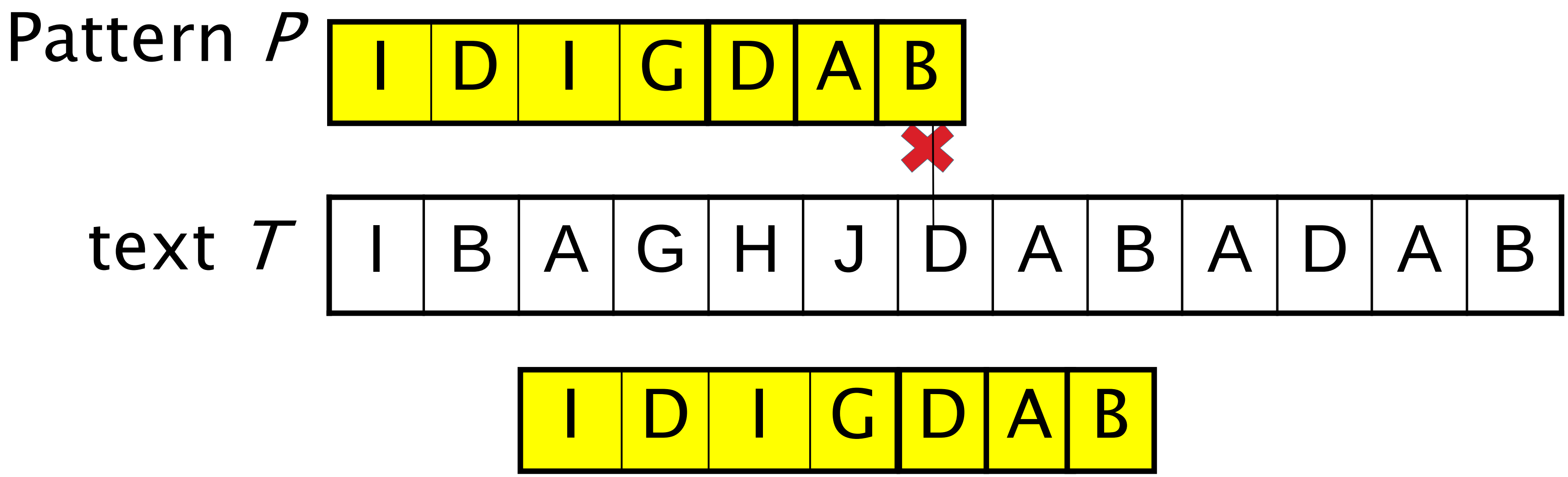
The Shift Table

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

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 ...



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”

Note: the algorithm is spelled out in detail in your textbook.

(it is called Horspool's algorithm)

Space-for-time tradeoffs varieties

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

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 ...

<u>Array:</u> 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

2 1D Array ...

1	A00043522
2	A00666666
3	
4	
	⋮
n-1	
n	

1	Bob
2	beelzebub
3	
4	
	⋮
n-1	
n	

1 2D Array ...

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

Using Sorted Array (2)

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

1	A00043522	Bob
2	A00066666	beelzebub
3	A00100000	186A0
4	A00111111	jimmy
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	Bob
2	A00066666	beelzebub
3	A00100000	186A0
4	A00111111	jimmy
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	Bob
2	A00066666	beelzebub
3		
4	A00100000	186A0
5	A00111111	jimmy
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	Bob
2	A00066666	beelzebub
3	A00099999	foo
4	A00100000	186A0
5	A00111111	jimmy
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)

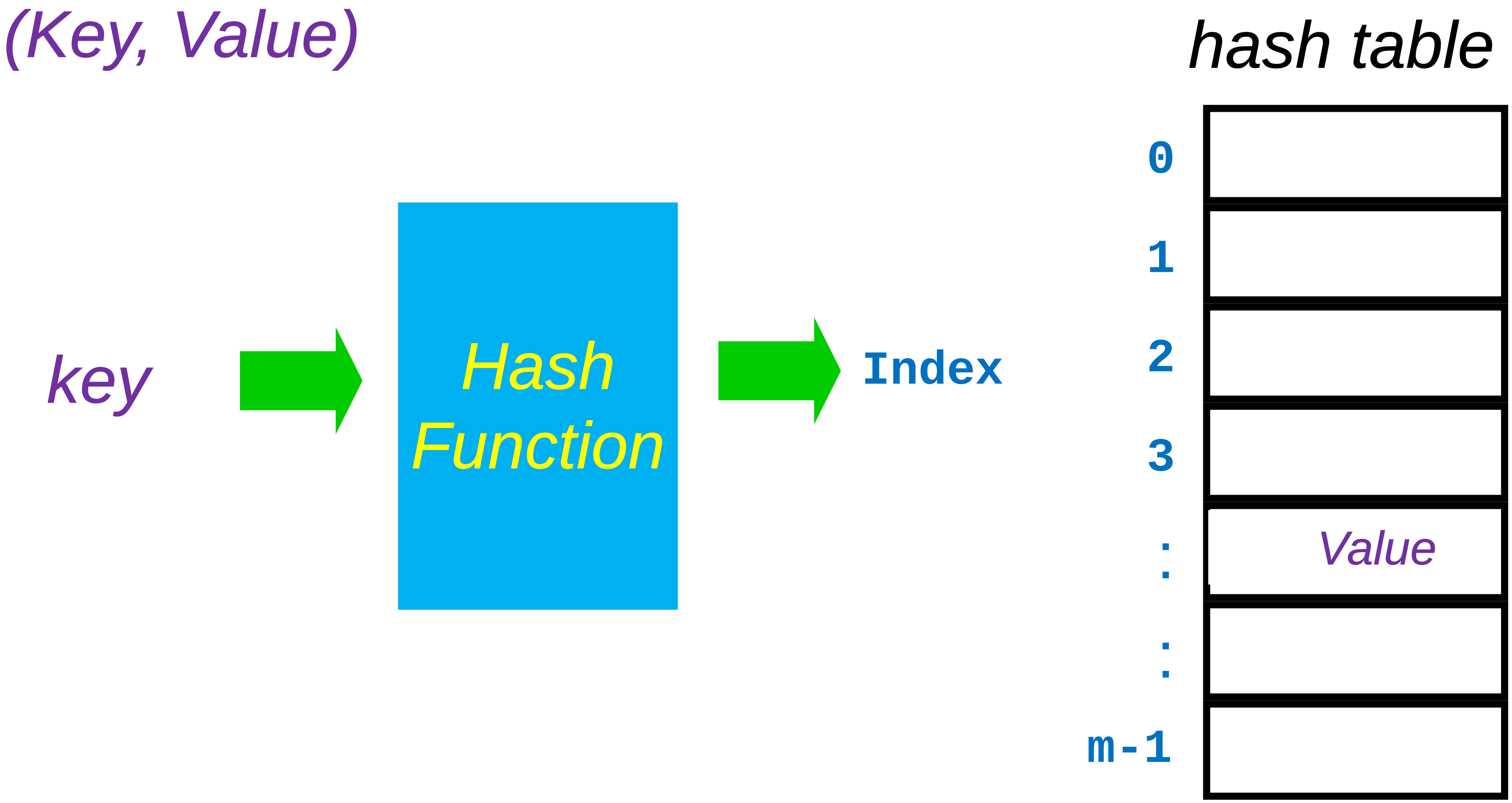
- *Search* operation is $O(logn)$
- *Retrieval* is $O(logn)$
- *Deletion* is $O(n)$

What if we use an unsorted Array:

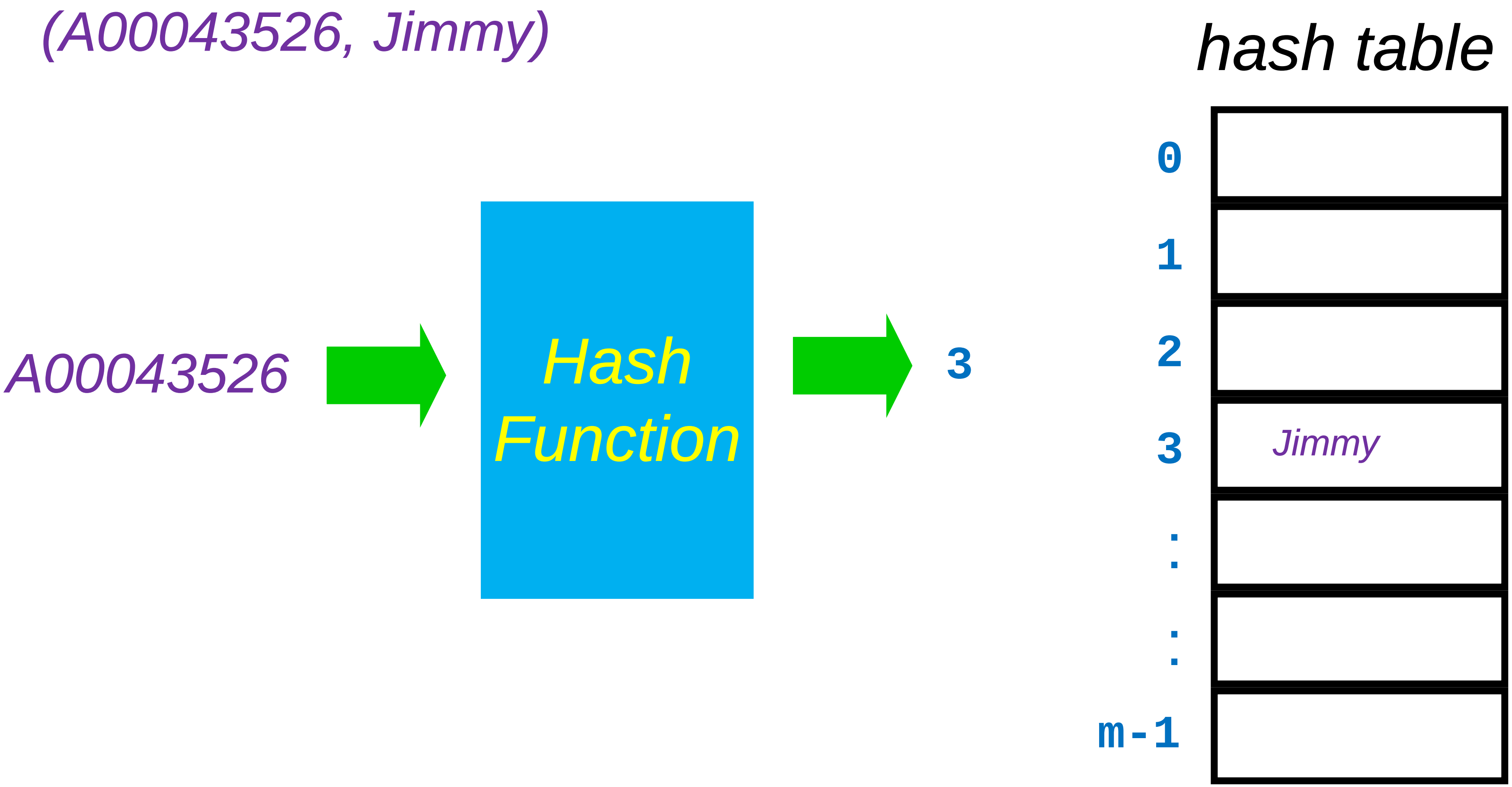
- *Insertion* will be much faster – O(1)
- *Searching, retrieve* will be slower – O(n)
- *Deletion* will be the same O(n)

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

Hashing/ Hash Table



Example


(A00043526, Jimmy)
hash table
A00043526
Hash Function
3
0
1
2
3
Jimmy
:
:
m-1

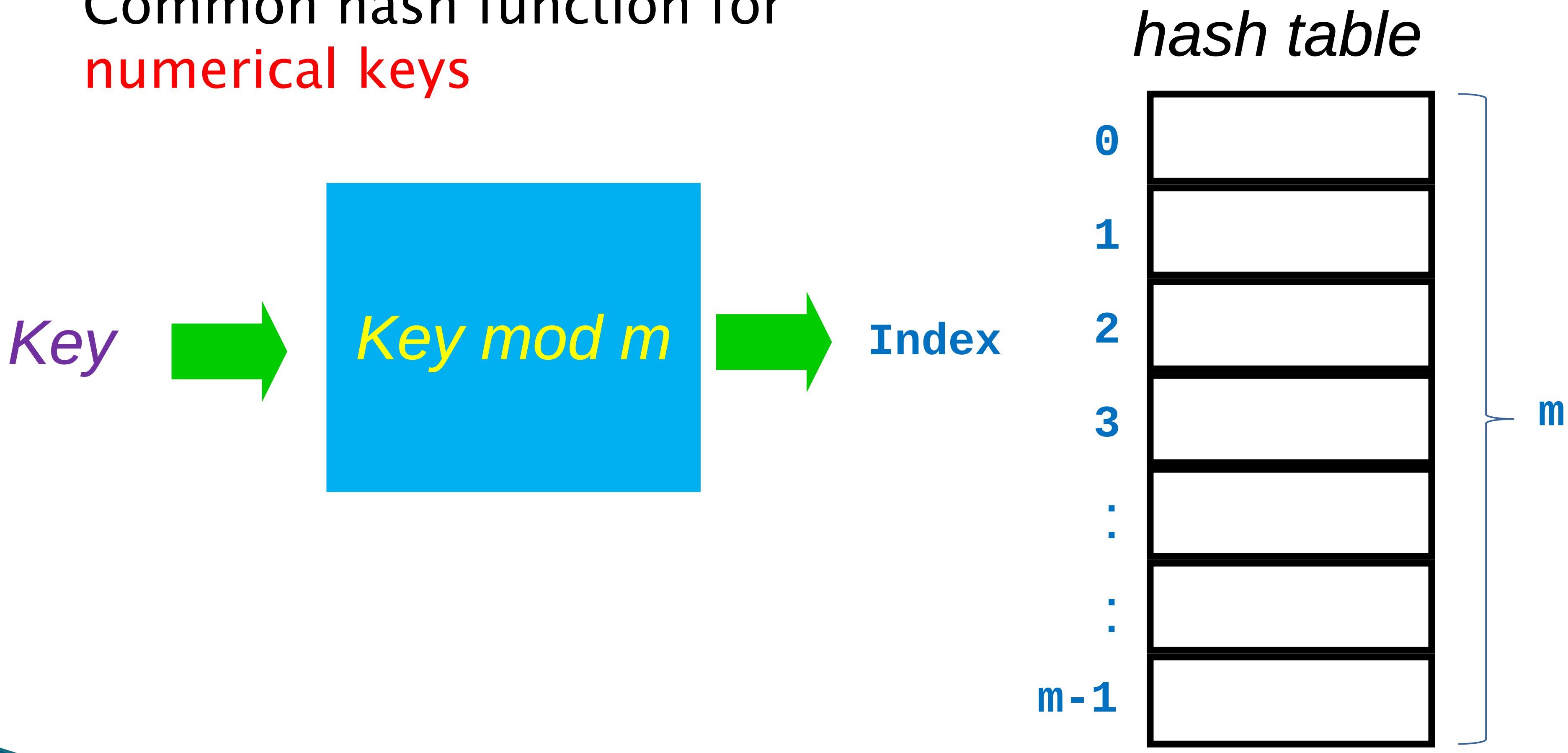
Hashing

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

f(key) = index

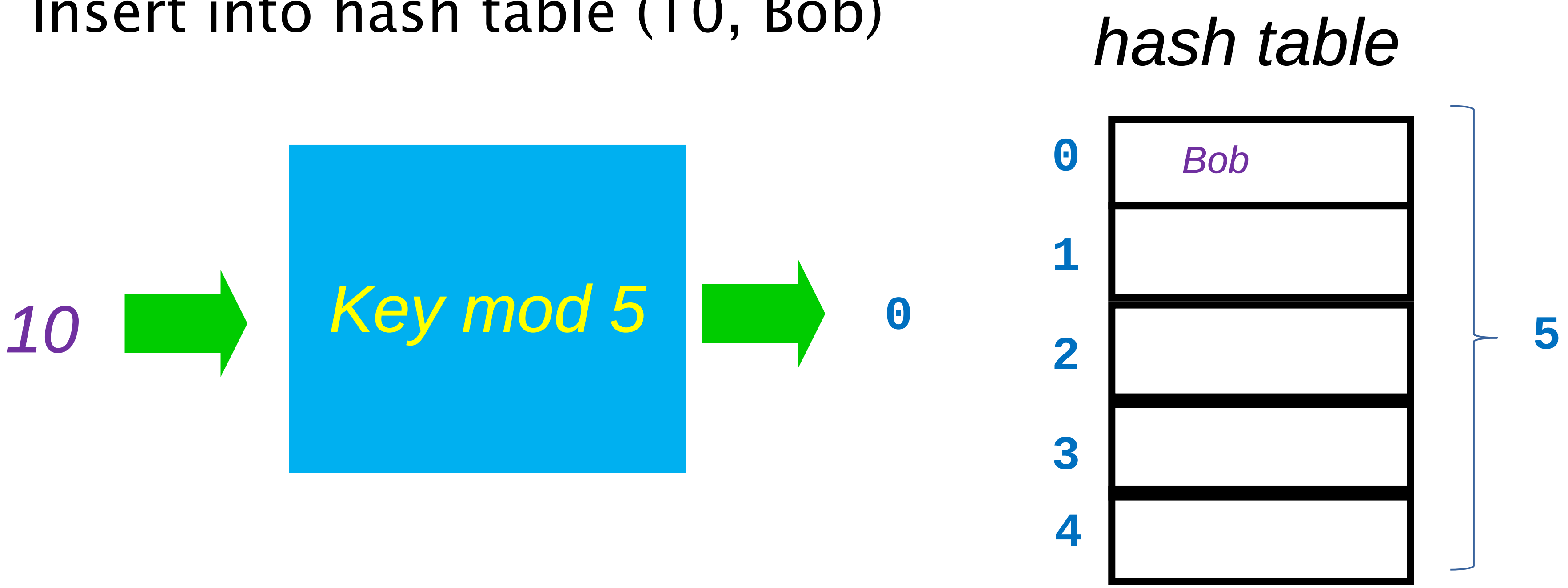
Hash Functions

Common hash function for
numerical keys



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

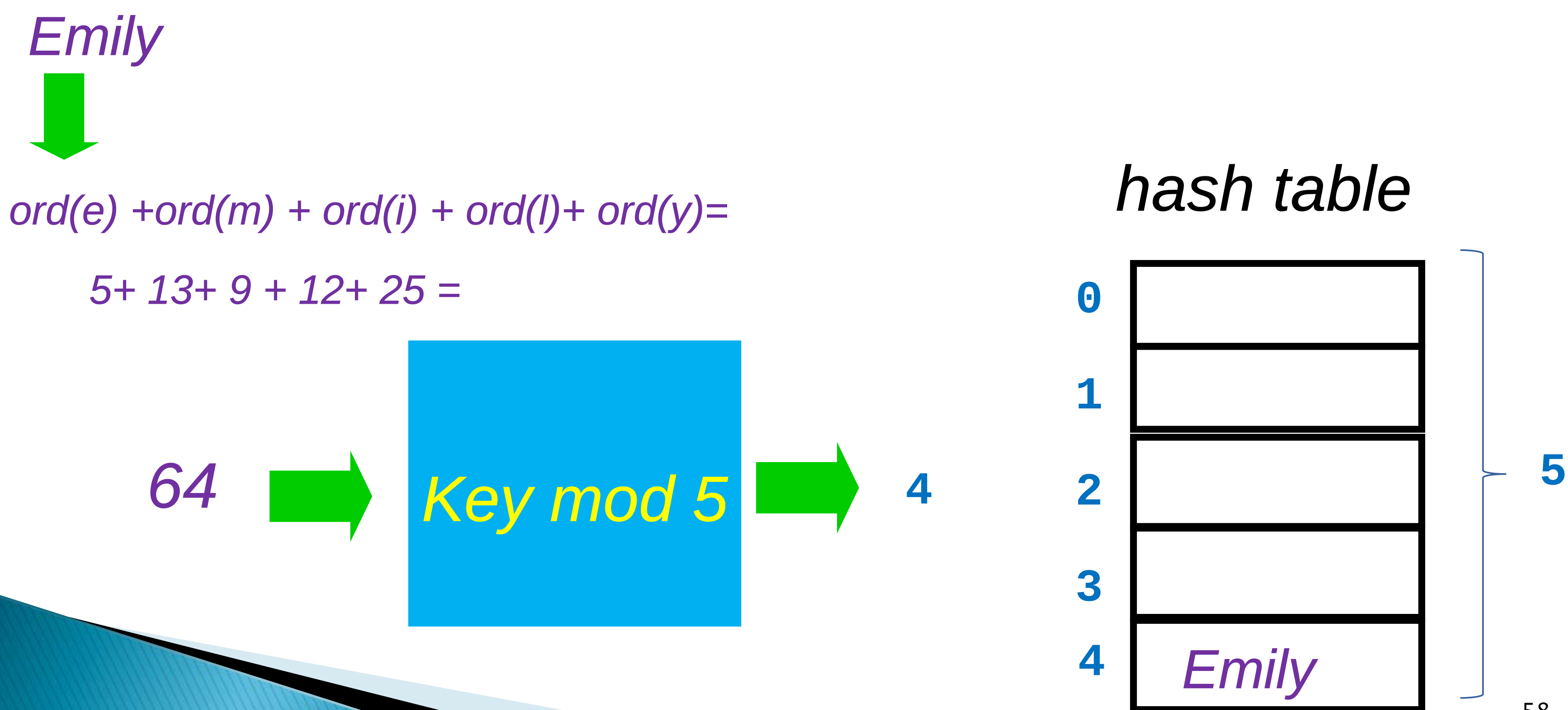
  Insert into hash table (Emily, 6046321)

Hash Functions

Example

assume m=5

Insert into hash table (Emily, 6046321)



Hash Functions

- Sample Hash function for the keys that are not number

```
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 i 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
```

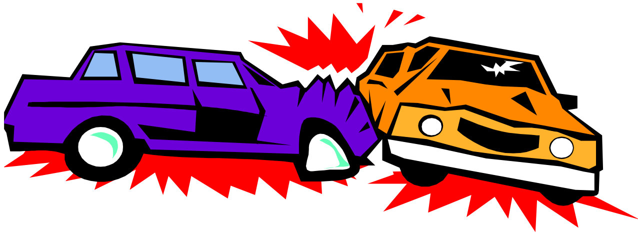
the actual hashcode depends on the number of buckets

Space-for-time tradeoffs varieties

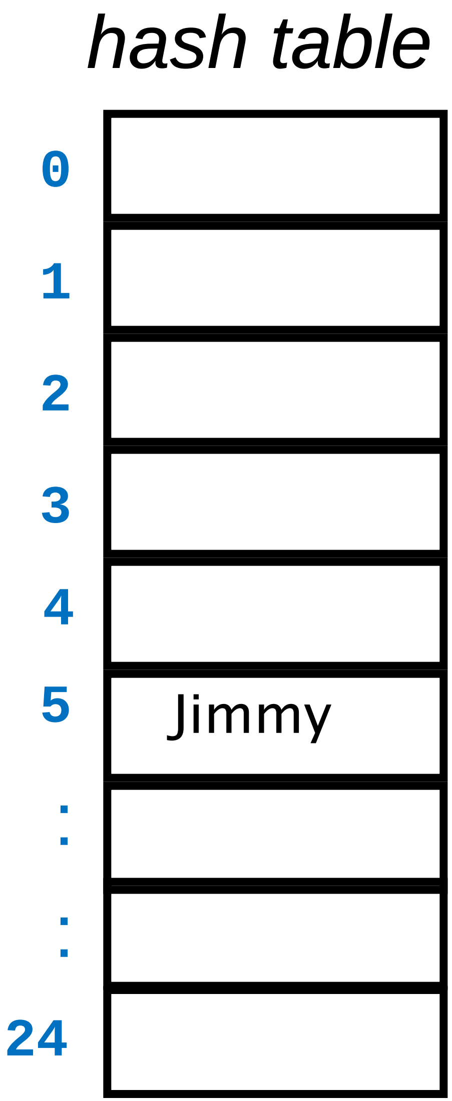
1. **Input enhancement:** preprocess the input to store some info to be used later in solving the problem
   - Comparison Counting Sort
   - Distribution Counting Sort
   - String Matching

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

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

Collisions Handling

Two way to handle collision:

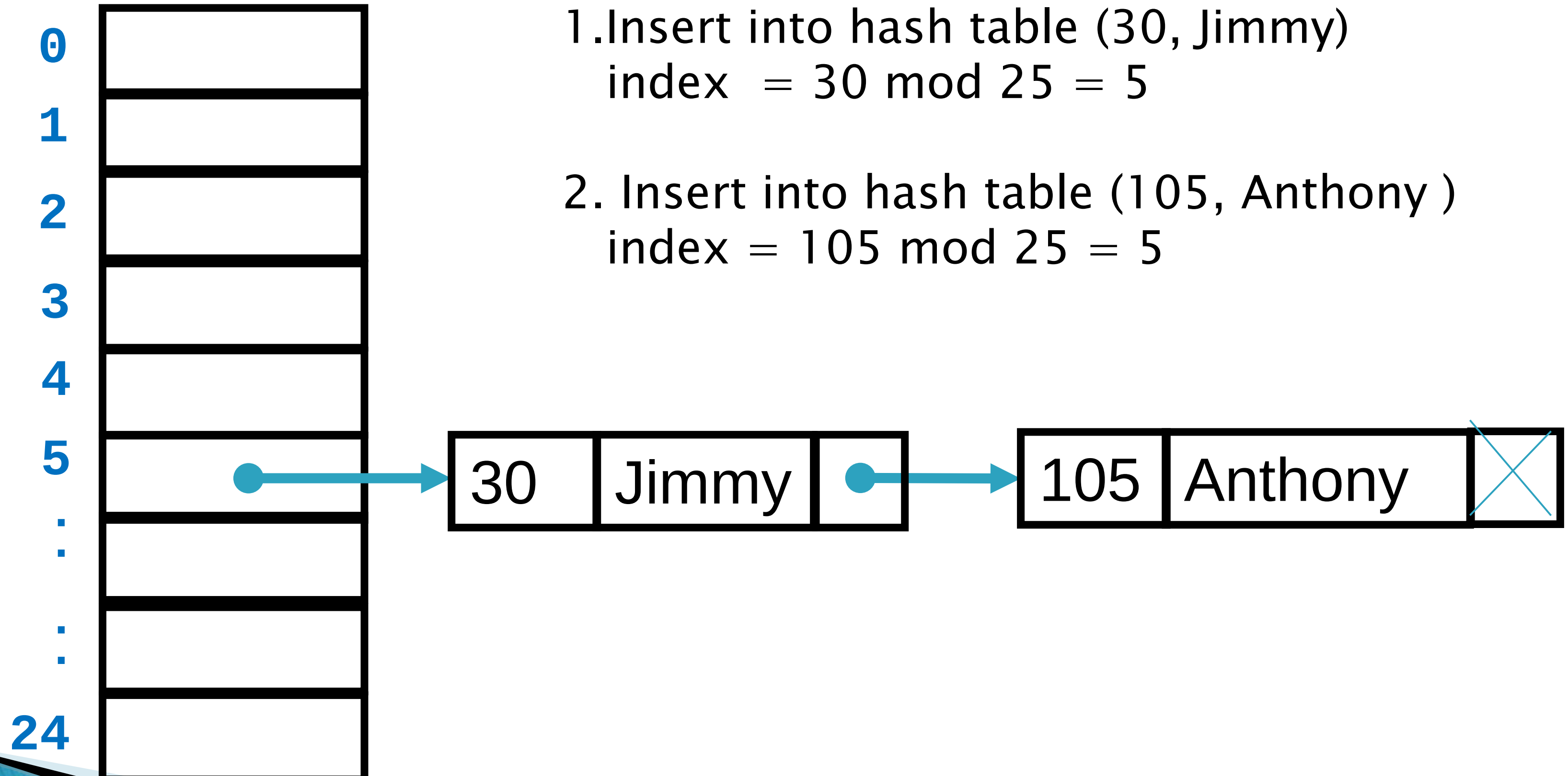
1. Separate Chaining
2. Closed Hashing

Collisions Handling –Separate Chaining

- Each bucket in the table point 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 changing Exercise 1

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

(44, name1)

(23, name2)

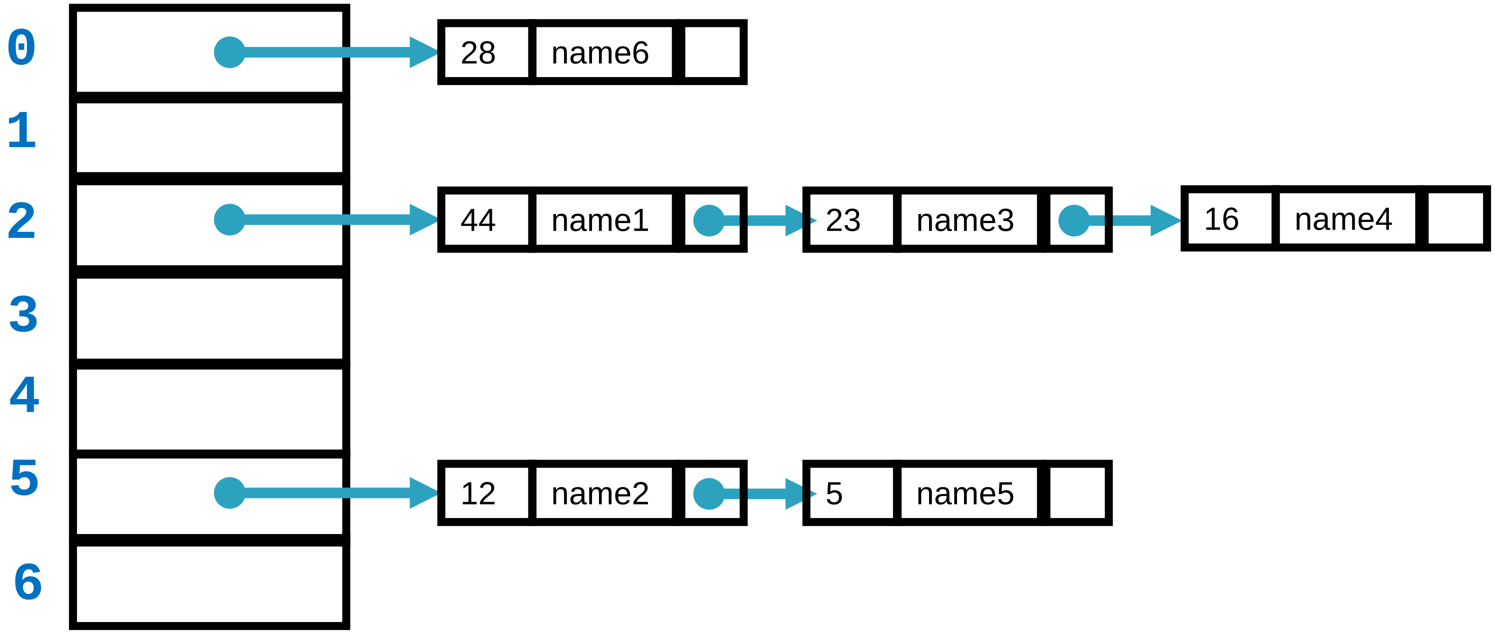
(16, name3)

(12, name4)

(5, name5)

(44, name1)
(12, name2)
(23, name3)
(16, name4)
(5, name5)
(28, name6)

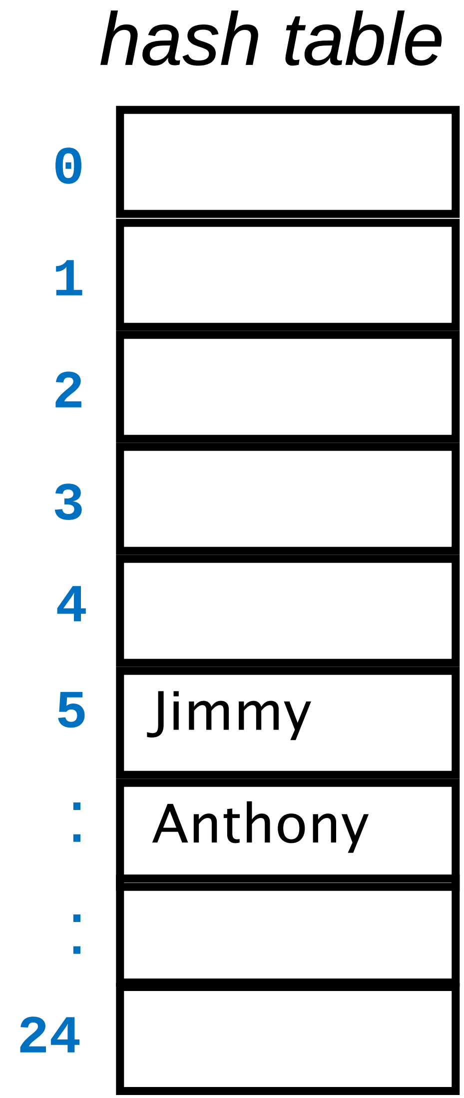
hash function h(i) = i mod 7



Closed Hashing

- 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!)

Closed Hashing



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, name1)

(12, name2)

(13, name3)

(88, name4)

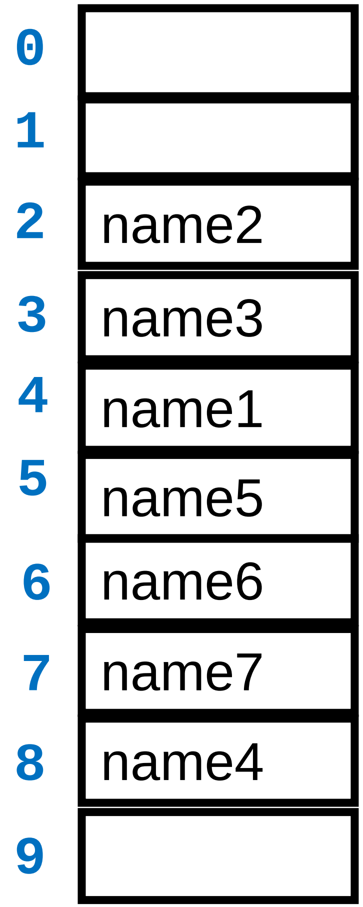
(23, name5)

(16, name6)

(22, name6)

(44, name1)
(12, name2)
(13, name3)
(88, name4)
(23, name5)
(16, name6)
(22, name7)

hash function h(i) = i mod 10



Space-for-time tradeoffs varieties

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

Efficiency of Hashing

What is the efficiency of the hashtable structure?

- **add**(key, value) ... is $O(1)$
- value ← **get**(key) ... is $O(1)$
- **delete**(key) ... is $O(1)$

◦ of course there could always be a degenerate case, where every insert causes a collision ... in this case we would end up with O(n)

→*conclusion : implementation of the hashing function is important*

→*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 algo (from the book):

```
h ← 0
for i ← 0 to s-1 do
   h ← h + ord(c_i)  // ord(c_i) is the relative 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 + (ascii(c_i) * alpha^(i))
code ← h mod numBuckets
```

  - Assuming alpha = 128 (number of ascii codes)
  - Assuming numbuckets = 99
    - dog = 64
    - god = 46
    - add = 26
    - dad = 65

Java's String.hashCode()

```
public int hashCode() {
  int h = 0;              // the final hashcode
  int off = 0;            // offset in to the string
  char val[] = value;     // put the string in an array of char
  int len = count;
  if (len < 16) {
     for (int i = len ; i > 0; i--) {
            h = (h * 37) + val[off++];
     }
  } else { // only sample some characters
     int skip = len / 8;
     for (int i=len ; i>0; i-=skip, off+=skip) {
            h = (h * 39) + val[off];
     }
  }
  return h;
}
```

Java's String.hashCode() (2)

- Java's hashcode() produces the following results ...
  - dog = 9
  - god = 90
  - add = 50
  - dad = 59

Try it/ homework

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

Hashing Exercise 3

- Devise an hash function to map the keys to buckets
- Draw a 10-element hashmap resulting from hashing of the keys using your hash function
- Use separate chaining for handling collision

```
a8s:elvis
se3:weasil
22a:pepper
14c:chili
aba:pretzel
1s1:elvis
d6e:angus
```

Hashing Exercise 3 (solution part 1)

One possible algorithm is similar to the one discussed earlier for strings, but we don't take the ordinal value for integers (ie: the char "4" is just assigned the integer value 4)

For example: the string c7 is Ord("c") + 7 = 3+7 = 10

Using this algorithm we get:

KEY VALUE	ORD SUM	HASHCODE
a8s:elvis	1+8+19=28	28 mod 10 = 8
se3:weasil	19+5+3=27	27 mod 10 = 7
22a:pepper	2+2+1=5	5 mod 10 = 5
14c:chili	1+4+3=8	8 mod 10 = 8
aba:pretzel	1+2+1=4	4 mod 10 = 4
1s1:elvis	1+19+1=21	21 mod 10 = 1
d6e:angus	4+6+5=15	15 mod 10 = 5

a=1
b=2
c=3
d=4
e=5
f=6
g=7
h=8
i=9
j=10
k=11
l=12
m=13
n=14
o=15
p=16
q=17
r=18
s=19
t=20
u=21
v=22
w=23
x=24
y=25
z=26

Hashing Exercise 3 (solution part 2)

- now we draw the hashmap
  - we will need to store the keys as well as the values ...