Data Structures and Algorithms

(CS210A)

Semester I – **2014-15**

Lecture 39

- **Integer sorting :** Radix Sort
- **Search data structure for integers :** Hashing

Types of sorting algorithms

In Place Sorting algorithm:

A sorting algorithm which uses only **O**(**1**) <u>extra space</u> to sort.

Example: Heap sort, Quick sort.

Stable Sorting algorithm:

A sorting algorithm which preserves the order of **equal keys** while sorting.

	0	1	2	3	4	5	6	7
A	**2**	**5**	**3**	**0**	**6.1**	**3**	**7.9**	**4**

	0	1	2	3	4	5	6	7
A	**0**	**2**	**3**	**3**	**4**	**5**	**6.1**	**7.9**

Example: Merge sort.

Integer Sorting algorithms

Continued from last class

Counting sort: algorithm for sorting integers

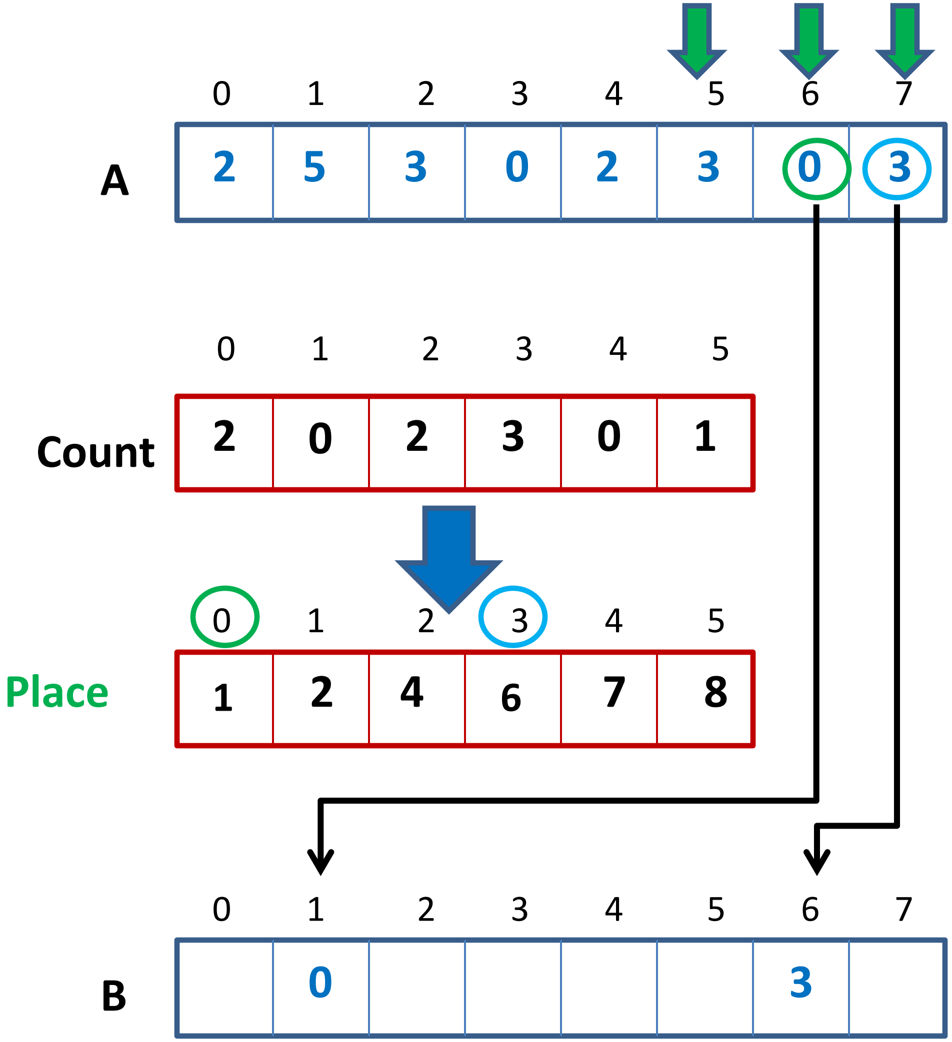
Input: An array **A** storing $\boldsymbol{n}$ integers in the range [$0...\boldsymbol{k}-\mathbf{1}$].

Output: Sorted array **A**.

Running time: **O**$(\boldsymbol{n}+\boldsymbol{k})$ in **word RAM** model of computation.

Extra space: **O**$(\boldsymbol{n}+\boldsymbol{k})$

Counting sort: a visual description



Why did we scan elements of **A** in reverse order (from index $n-1$ to 0) while placing them in the final sorted array **B** ?

Answer:

- To ensure that Counting sort is **stable**.
- The reason why stability is required will
- become clear soon ☺

Counting sort: algorithm for sorting integers

CountSort(A[$0 \ldots n-1$], k)

For j=0 **to** $k-1$ **do** Count[j]← 0;

For i=0 **to** $n-1$ **do** Count[A[i]]← Count[A[i]] +1;

For j=0 **to** $k-1$ **do** Place[j]← Count[j];

For j=1 **to** $k-1$ **do** Place[j]← Place[$j-1$] + Count[j];

For i=$n-1$ to 0 **do**

{ B[Place[A[i]]-1]← A[i];

Place[A[i]] ← Place[A[i]]-1;

}

return B;

Counting sort: algorithm for sorting integers

Key points of Counting sort:

- It performs arithmetic operations involving **O**(**log** n + **log** k) bits

  (**O**(1) time in **word RAM**).
- It is a **stable** sorting algorithm.

Theorem: An array storing n integers in the range $[0..k-1]$can be sorted in **O**(n+k) time and using total **O**(n+k) space in **word RAM** model.

➔ **For** k **≤** n, we get an optimal algorithm for sorting.

➔ For $k = n^t$, time and space complexity is **O**(n^t).

(too bad for $t > 1$. ☹)

Question:

How to sort n integers in the range $[0..n^t]$ in **O**(tn) time and using **O**(n) space?

Radix Sort

Digits of an integer

$$507266$$

No. of **digits** = 6

value of **digit** $\in \{0, \ldots, 9\}$

$$\overbrace{1011}\overbrace{0001}\overbrace{0101}\overbrace{1111}$$

No. of **digits** = 4

value of **digit** $\in \{0, \ldots, 15\}$

It is up to us how we define digit ?

Radix Sort

Input: An array **A** storing n integers, where

(i) each integer has exactly d **digits**.

(ii) each **digit** has **value** $< k$

(iii) $k < n$.

Output: Sorted array **A**.

Running time:

O(dn) in **word RAM** model of computation.

Extra space:

O($n + k$)

Important points:

- makes use of a **count sort.**
- Heavily relies on the fact that **count sort** is a **stable sort** algorithm.

Demonstration of Radix Sort through example

A

2	0	1	2
1	3	8	5
4	9	6	1
5	8	1	0
2	3	7	3
6	2	3	9
9	6	2	4
8	2	9	9
3	4	6	5
7	0	9	8
5	5	0	1
9	2	5	8

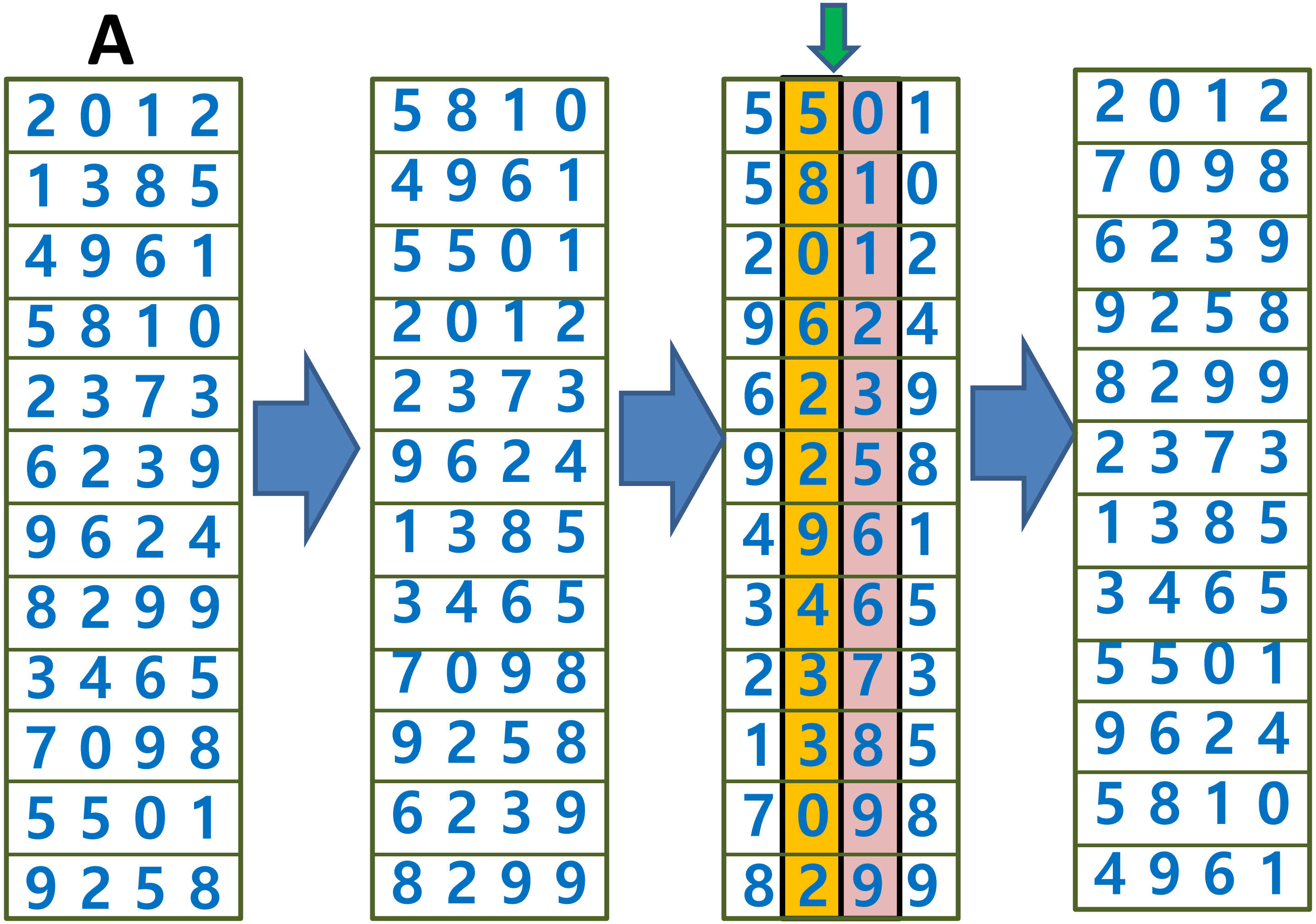
5	8	1	0
4	9	6	1
5	5	0	1
2	0	1	2
2	3	7	3
9	6	2	4
1	3	8	5
3	4	6	5
7	0	9	8
9	2	5	8
6	2	3	9
8	2	9	9

d =4
n =12
k =10

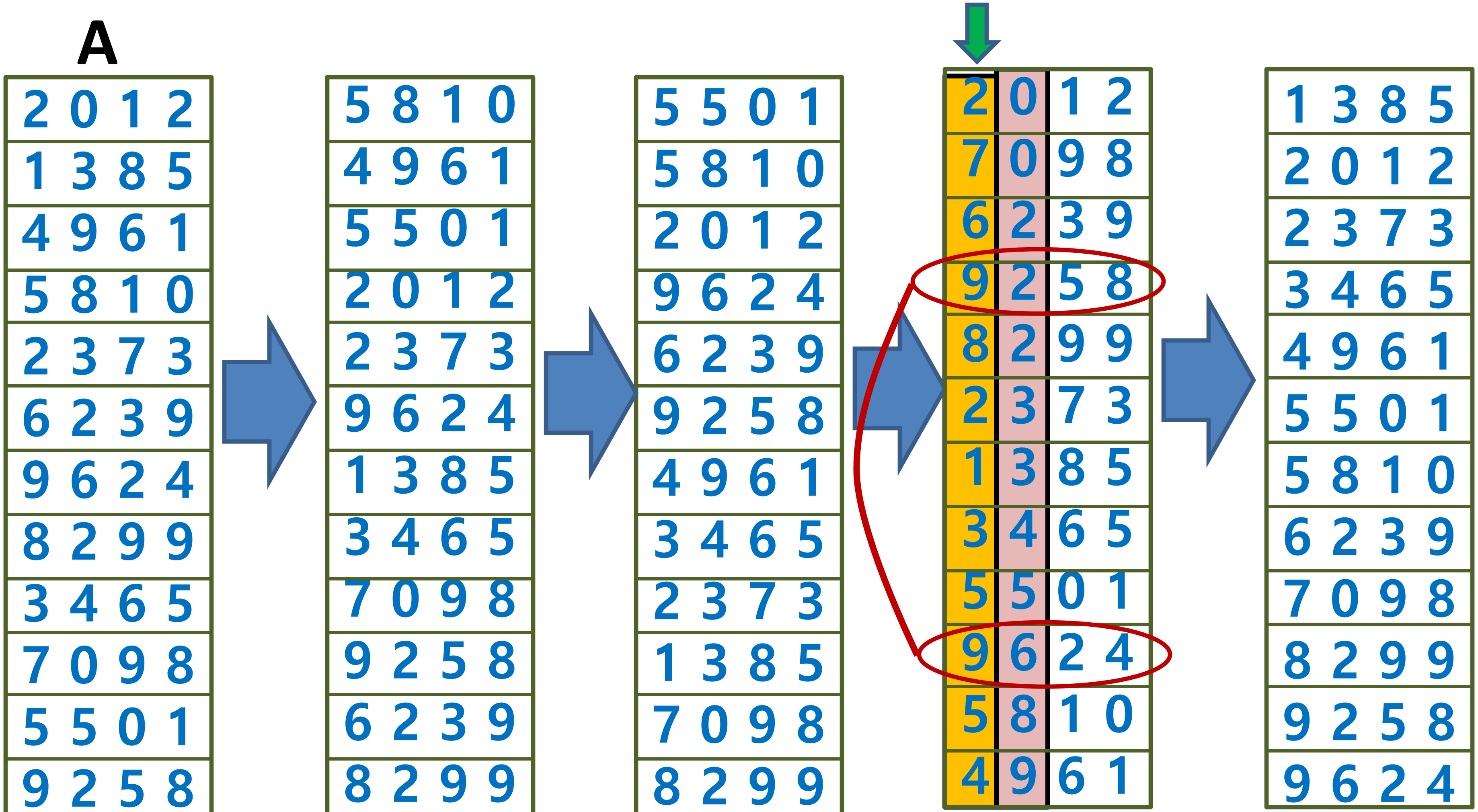
Demonstration of Radix Sort through example

A		
2 0 1 2	5 8 1 0	5 5 0 1
1 3 8 5	4 9 6 1	5 8 1 0
4 9 6 1	5 5 0 1	2 0 1 2
5 8 1 0	2 0 1 2	9 6 2 4
2 3 7 3	2 3 7 3	6 2 3 9
6 2 3 9	9 6 2 4	9 2 5 8
9 6 2 4	1 3 8 5	4 9 6 1
8 2 9 9	3 4 6 5	3 4 6 5
3 4 6 5	7 0 9 8	2 3 7 3
7 0 9 8	9 2 5 8	1 3 8 5
5 5 0 1	6 2 3 9	7 0 9 8
9 2 5 8	8 2 9 9	8 2 9 9

Demonstration of Radix Sort through example



Demonstration of Radix Sort through example



Can you see where we are exploiting the fact that **Countsort** is a **stable** sorting algorithm ?

Radix Sort

RadixSort(**A**[$0 \ldots n-1$], d, k)

{   **For** j=1 **to** d **do**

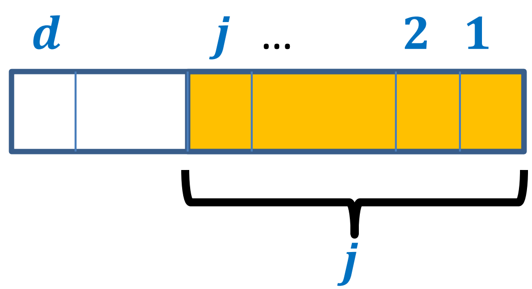
        Execute CountSort(**A**,k) with j**th digit** as the **key**;

    return **A**;

}

Correctness:

A number stored in **A** ➔



Inductive assertion:

At the end of j**th** iteration, array **A** is sorted according to the last j digits.

During the induction step, you will have to use the fact that **Countsort** is a **stable** sorting algorithm.

Radix Sort

RadixSort(A[$0 \ldots n-1$], d, k)

{ **For j=1 to d do**

    Execute CountSort(A,k) with **jth digit** as the **key;**

  return **A;**

}

Time complexity:

- A single execution of **CountSort(A,k)** runs in **O**($n+k$) **time** and **O**($n+k$) **space**.
- Note $k < n$,

  ➔ a single execution of **CountSort(A,k)** runs in **O**(n) time.

  ➔ Time complexity of radix sort = **O**(dn).
- ➔Extra space used = **O**(n)

Question: How to use Radix sort to sort n integers in range $[0..n^t]$ in **O**(tn) time and **O**(n) space ?

Answer:

1 bit

log n bits

d	k	Time complexity
$t \log n$	2	**O**($tn \log n$)
t	n	**O**(tn)

What digit to use ?

Power of the word RAM model

- **Very fast** algorithms for **sorting integers:**

  Example: n integers in range $[0..n^{10}]$ in **O**(n) time and **O**(n) space ?

- **Lesson:**

  Do not always go after **Merge sort** and **Quick sort** when input is integers.

- **Interesting programming exercise** (for winter vacation)**:**

  Compare Quick sort with **Radix sort** for sorting long integers.

Data structures for searching

in O(1) time

Motivating Example

Input: a given set S **of 1009 positive integers**

Aim: Data structure for **searching**

Example

{

123, 579236, 1072664, 770832456778, 61784523, 100004503210023, $\boxed{19}$,
762354723763099, 579, 72664, $\boxed{977083245677001238}$, 84, 100004503210023,

…

}

Data structure : Array storing S in sorted order

Searching : Binary search

O(log $|S|$) time

Can we perform search in O(1) time ?

Problem Description

$\boldsymbol{U}$: $\{0,1, \ldots, \boldsymbol{m}-\mathbf{1}\}$ called **universe**

$\boldsymbol{S} \subseteq \boldsymbol{U}$,

$\boldsymbol{n} = |\boldsymbol{S}|$,

$\boldsymbol{n} \ll \boldsymbol{m}$

Aim: A data structure for a given set $\boldsymbol{S}$ that can facilitate searching in **O(1)** time.

A search query: Does $\boldsymbol{j} \in \boldsymbol{S}$?

Note: $\boldsymbol{j}$ can be any element from $\boldsymbol{U}$.

A trivial data structure for O(1) search time

Build a 0-1 array **A** of size m such that

A[i] = 1 **if** $i \in S$.

A[i] = 0 **if** $i \notin S$.

Time complexity for searching an element in set S **: O(1).**

	0	1	2	3	4	…				…			m-1
A	0	0	1	0	0	…	0	1	0	…	0	0	1

This is a totally Impractical data structure because $n \ll m$!

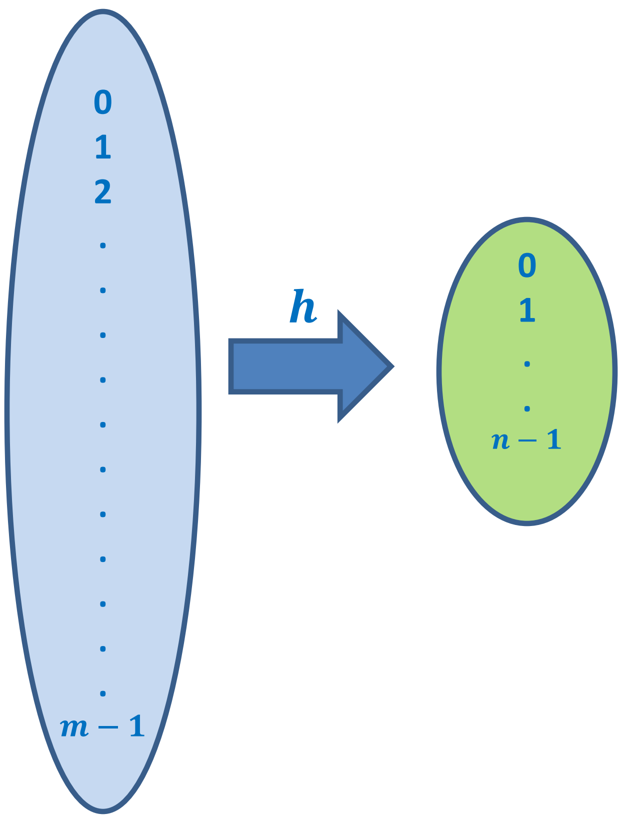
Example: n = few thousands, m = few trillions.

Question:

Can we have a data structure of **O**(n) size that can answer a search query in **O**(**1**) time ?

Answer: **Hashing**

Hash function, hash value



Hash function:

h is a mapping from U to $\{0,1,\ldots,n-1\}$ with the following characteristics.

- **Space** required for h : a few **words**.
- $h(i)$ computable in **O(1) time** in **word RAM.**

Example: $h(i)$ = i **mod** n

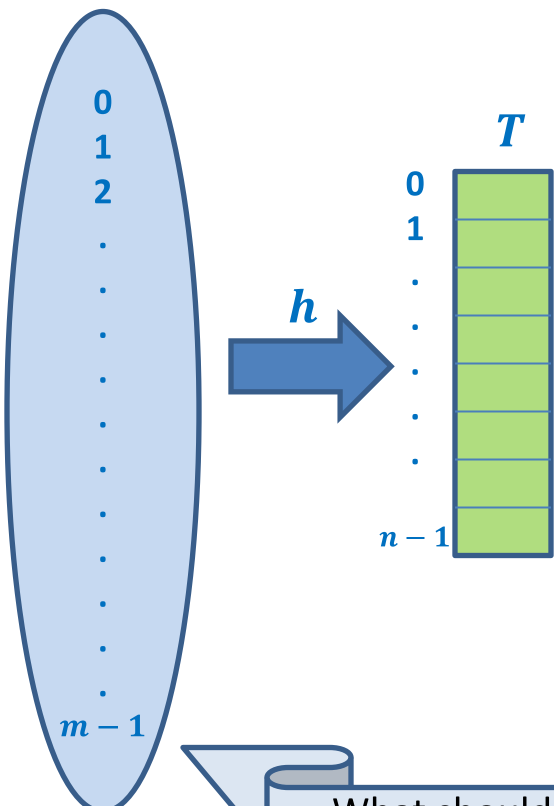
Hash value:

$h(i)$ is called hash value of i for a given hash function h, and $i \in U$.

Hash Table:

An array $T[0 \ldots n-1]$ of ...

Hash function, hash value, hash table



What should T store?
Ponder over it…
We shall discuss it in the next class.

Hash function:

h is a mapping from U to $\{0,1,\ldots,n-1\}$ with the following characteristics.

- **Space** required for h : a few **words**.
- $h(i)$ computable in **O(1) time** in **word RAM.**

Example: $h(i)$ = i **mod** n

Hash value:

$h(i)$ is called hash value of i for a given hash function h, and $i \in U$.

Hash Table:

An array $T[0 \ldots n-1]$ of …