# Data Structures and Algorithms

(CS210A)

## Lecture 40

- Search data structure for integers : Hashing
- Quick sort : some facts

# 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

$\boldsymbol{U}$ :

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

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

**A search query:**

**Aim: A data structure** for a <u>given</u> set $\boldsymbol{S}$

that can facilitate search in **O**($\mathbf{1}$) time

# 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 $n \ll m$ !**

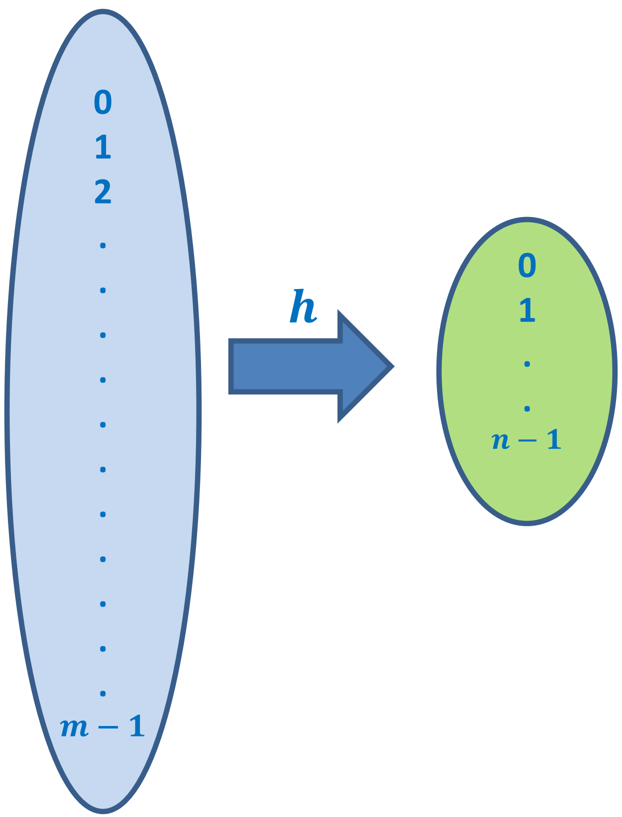
**Example: $n$ = few thousands, $m$ = few trillions.**

**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 function:**

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

- **Space** required for $h$
- $h(i)$ computable in

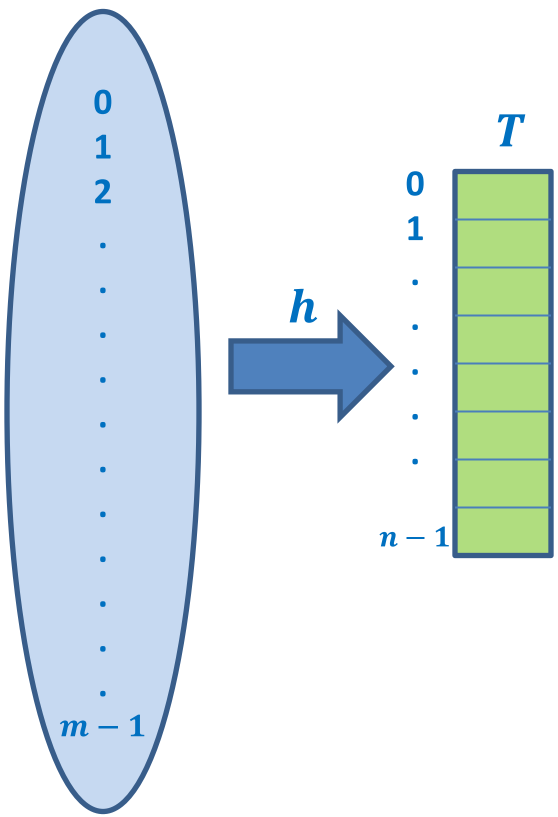
**Example:**

**Hash value:**

For a given hash function $h$, and $i \in U$.

$h(i)$ is called hash value of $i$

# 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** $n$

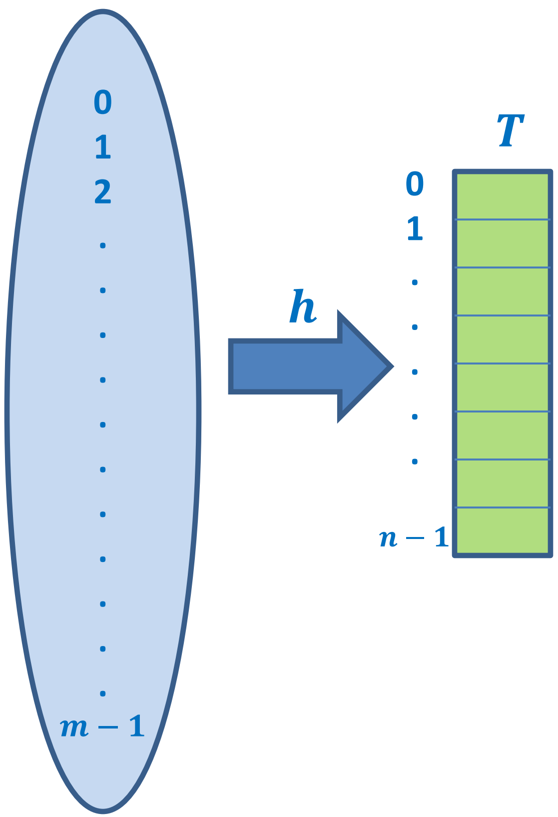
**Hash value:**

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

**Hash Table:**

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

# Hash function, hash value, hash table




**Hash function:**

$h$ is a mapping from $U$ to $\{0,1,\dots,n-1\}$ 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:**

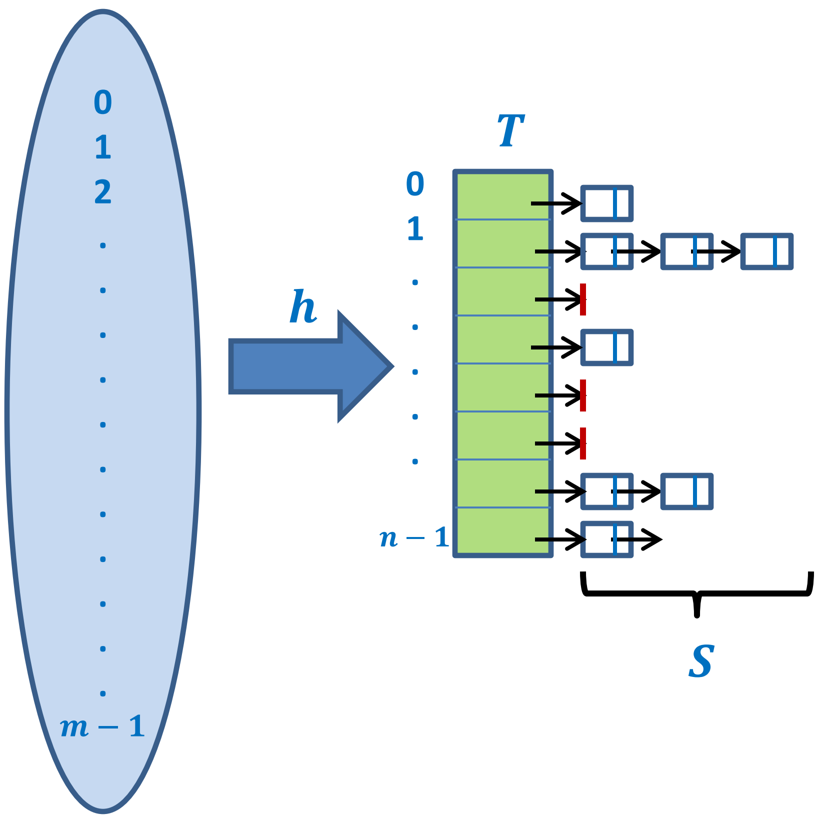
For a given hash function $h$, and $i \in U$.

$h(i)$ is called hash value of $i$

**Hash Table:**

An array $T[0 \dots n-1]$

# Hash function, hash value, hash table




**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:**

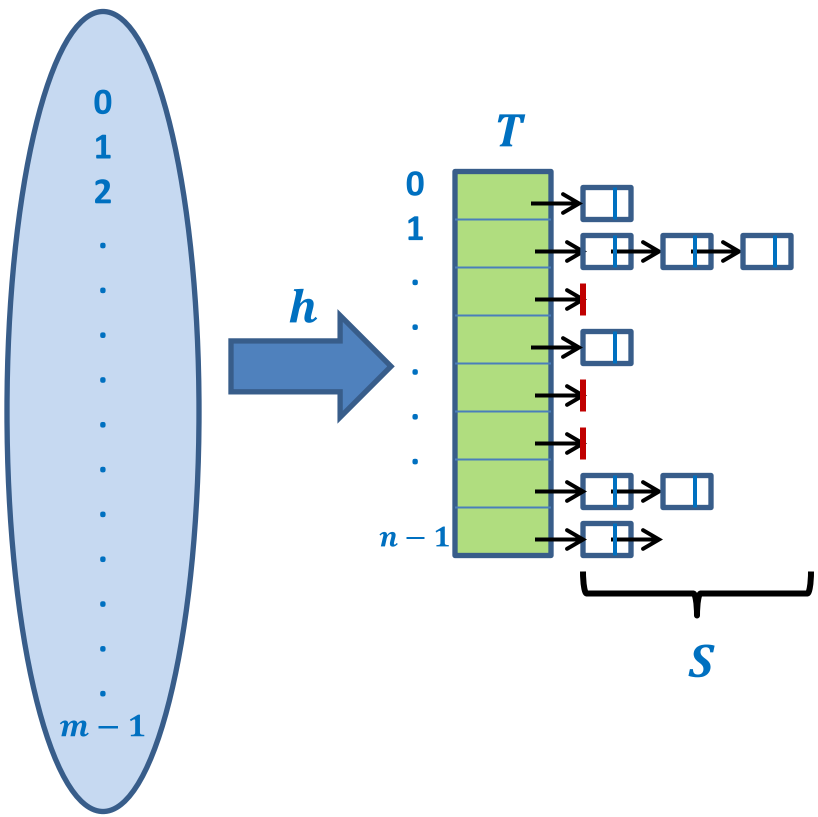
For a given hash function $h$, and $i \in U$.

$h(i)$ is called hash value of $i$

**Hash Table:**

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

# Hash function, hash value, hash table




**Question:**

How to use ($h$,$T$) for searching an element $i \in U$?

**Answer:**

$k \leftarrow h(i)$;

Search element $i$ in the list $T[k]$.

**Time complexity for searching:**

**O**(length of the **longest** list in $T$).

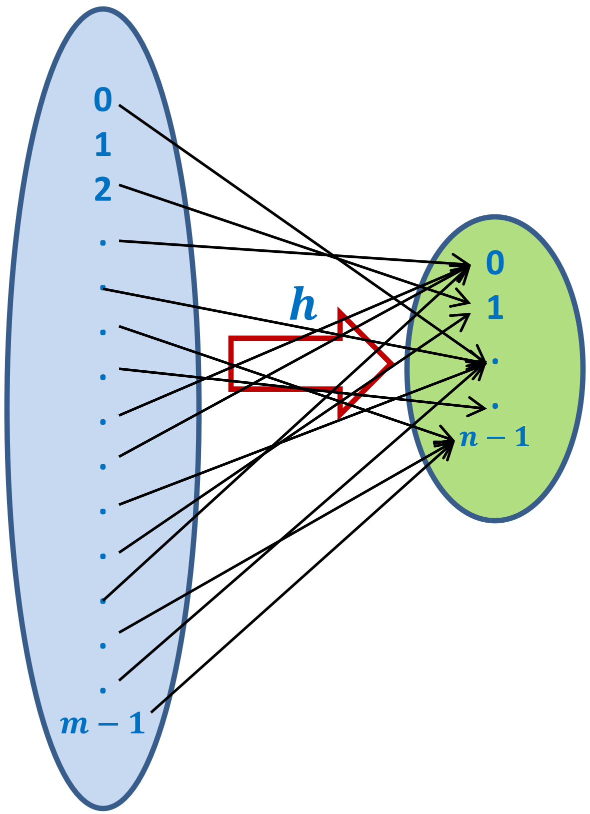
# Efficiency of Hashing depends upon hash function

A hash function $h$ is <u>good</u> if it can **evenly** distributes $S$.

**Aim:** To search for a good hash function for a given set $S$.


There **can not be** any hash function $h$ which is good for **<u>every</u>** $S$**.**

# Hash function, hash value, hash table




For every $h$, there exists a subset of $\left\lceil \frac{m}{n} \right\rceil$ elements from $U$ which are hashed to same value under $h$.

So we can always construct a subset $S$ for which all elements have same hash value

➔ All elements of this set $S$ are present in a single list of the hah table $T$ associated with $h$.

➔ **O**($n$) worst case search time.

# Why does hashing work so well in Practice ?

$$\boldsymbol{h}(i) = i \ \mathbf{mod} \ n$$

Because the set $\boldsymbol{S}$ is usually a **uniformly random** subset of $\boldsymbol{U}$.

Let us do a **theoretical analysis**
**to prove** this fact.

# Why does hashing work so well in Practice ?

$\boldsymbol{h}(x) = \boldsymbol{x} \ \mathbf{mod}\ n$

Let $y_1, y_2, \ldots, y_n$ denote $n$ elements selected <u>randomly uniformly</u> from $\boldsymbol{U}$ to form $\boldsymbol{S}$.

**Question:**

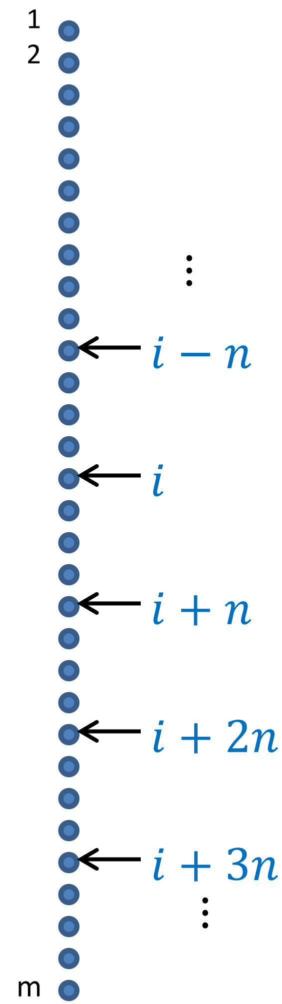
What is expected number of elements of $\boldsymbol{S}$ colliding with $y_1$?

**Answer:** Let $y_1$ takes value $i$.

**P**($y_j$ collides with $y_1$) = ??

How many possible values can $y_j$ take ? $m-1$

How many possible values can collide with $i$ ?



# Why does hashing work so well in Practice ?

$\boldsymbol{h}(x) = \boldsymbol{x} \ \mathbf{mod} \ n$

Let $y_1, y_2, \ldots, y_n$ denote $n$ elements selected <u>randomly uniformly</u> from $\boldsymbol{U}$ to form $\boldsymbol{S}$.

**Question:**

What is expected number of elements of $\boldsymbol{S}$ colliding with $y_1$?

**Answer:** Let $y_1$ takes value $i$.

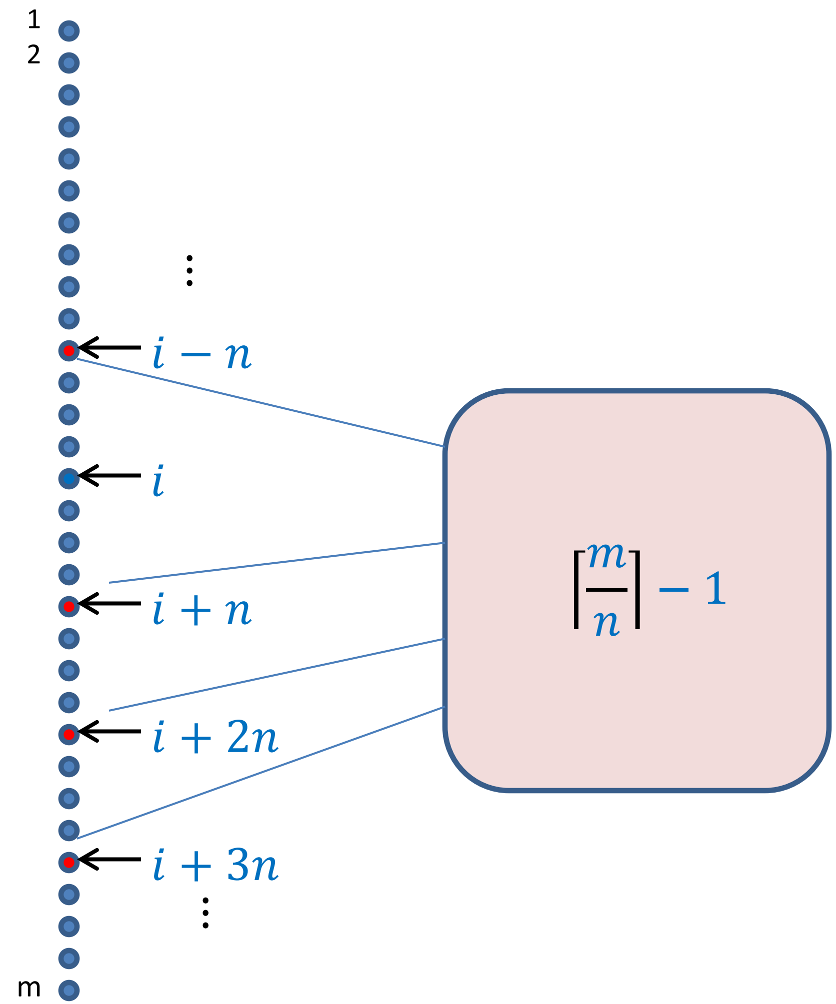
**P**($y_j$ collides with $y_1$) =

$$\frac{\lceil \frac{m}{n} \rceil - 1}{m-1}$$

Expected number of elements of $\boldsymbol{S}$ colliding with $y_1$ =

$$= \frac{\lceil \frac{m}{n} \rceil - 1}{m-1} (n-1)$$

$$= O(1)$$



# Why does hashing work so well in Practice ?

**Conclusion**

1. $\boldsymbol{h}(i) = i \ \mathbf{mod} \ n$ works so well because
for a uniformly random subset of $\boldsymbol{U}$,
the **expected** number of collision at an index of $\boldsymbol{T}$ is **O**(1).

It is easy to fool this hash function such that it achieves **O**($s$) search time.
(do it as a simple exercise).

**This makes us think:**

"How can we achieve worst case **O**(1) search time for a given set $\boldsymbol{S}$."

# Hashing: theory

1953

1984

$\boldsymbol{U}$ : $\{0,1,\ldots,\boldsymbol{m}-\boldsymbol{1}\}$

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

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

**Theorem [FKS, 1984]:**

A hash table and hash function can be computed

**Space** : **O**($\boldsymbol{n}$)

**Query time**: worst case **O**($\boldsymbol{1}$)

How complicated would it be ?

**Ingredients :**

- elementary knowledge of **prime numbers**.
- The algorithms use **simple randomization**.

**(We shall discuss such an algorithm in CS345.)**

# Quick Sort

## Facts

**(invented by Tony Hoare in 1960)**

# Quick sort versus Merge Sort

| | Merge Sort | Quick Sort |
|---|---|---|
| Average case comparisons | $n \log_2 n$ | $1.39\, n \log_2 n$ |
| Worst case comparisons | $n \log_2 n$ | $n(n-1)$ |

**Realization from Programming assignment 4 (part 1):**

| | $n = 100$ | $n = 1000$ | $n \geq 10000$ |
|---|---|---|---|
| No. of times **Merge sort** outperformed **Quick sort** | $0.1\%$ | $0.02\%$ | $0\%$ |

**Reasons** :

- Overhead of **Copying** in merging **?**
- **Technical (cache)**

Very few students tried to find out ☹

# What makes Quick sort popular ?

No. of repetitions = $1000$

| No. of times run time exceeds average by | 100 | 1000 | $10^4$ | $10^5$ | $10^6$ |
|---|---|---|---|---|---|
| $10\%$ | 190 | 49 | 22 | 10 | 3 |
| $20\%$ | 28 | 17 | 12 | 3 | 0 |
| $50\%$ | 2 | 1 | 1 | 0 | 0 |
| $100\%$ | 0 | 0 | 0 | 0 | 0 |

**Inference**:

The chances of deviation from average case

➔ The *reliability* of quick sort

Can this behavior of Quick sort be explained **theoretically** ?

# What makes Quick sort popular ?

**Theorem** [Colin McDiarmid, **1991**]:

Prob. the run time exceeds average by $x\%$ = $n^{-\frac{x}{100}\ln\ln n}$

➔

Prob. run time is double the average for $n = 10^6$ is $10^{-15}$

Prob. any (**INTEL/AMD/ …**) **CPU** failure in 720 hours is $0.05$

Isn't it amazing that some of you still don't rely upon Quick sort!☺

Refer to the following paper (at least read the abstract):

**Title:** Cycles, Cells and Platters: An Empirical Analysis of Hardware Failures on a Million Consumer PCs

**Authors**: Edmund B. Nightingale, John R. Douceur, Vince Orgovan

**Available at** : research.microsoft.com/pubs/144888/eurosys84-nightingale.pdf

or just **google** the title

# But a serious problem with Quick sort.

- **Distribution sensitive** ☹
- Can be fooled easily
  - sort in increasing order
  - Sort in decreasing order

**Solution**:

Select pivot element **randomly uniformly** in each call

This is **randomized quick sort**.