Data Structures and Algorithms

(CS210A)

Semester I – **2014-15**

Lecture 40

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

Data structures for searching

in $O(1)$ time

Problem Description

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

$S \subseteq U$,

$n = |S|$, $n \ll m$

A search query: Given any $j \in U$, is j **present in** S ?

Aim: A data structure for a given set S that can facilitate search in **O**(1) time in $\boxed{\textbf{word RAM}}$ model.

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 $n \ll m$!

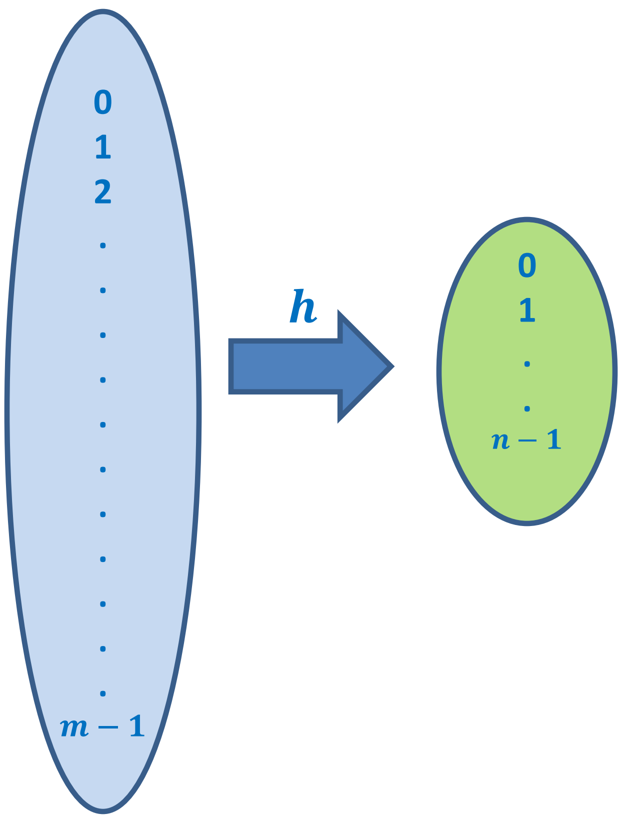
Example: n = few thousands, m = few trillions.

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

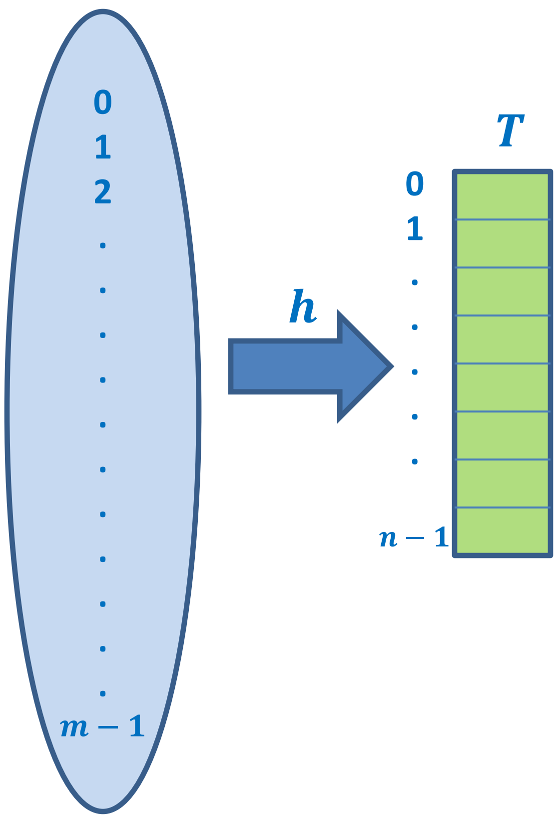
Hash value:

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

Hash Table:

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

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

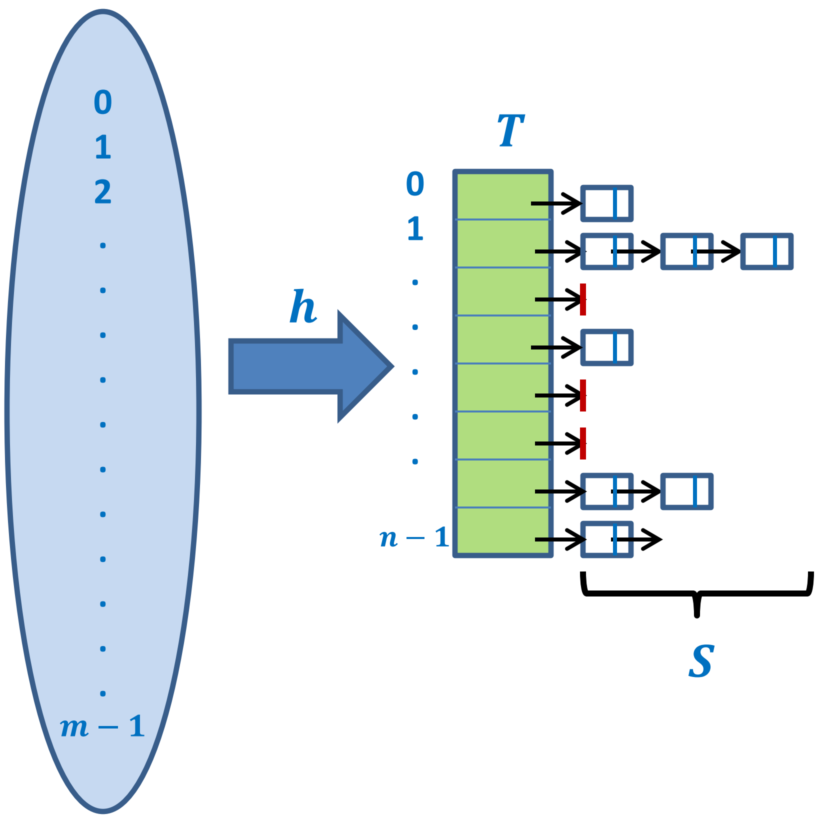
Hash value:

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

Hash Table:

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

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

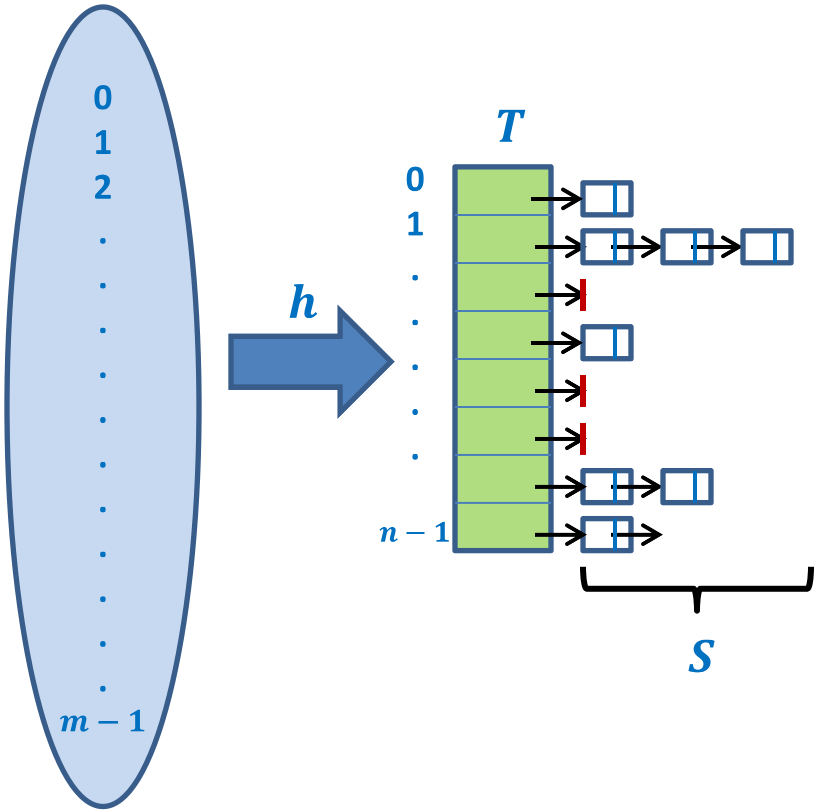
Hash value:

$h(i)$ 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 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).

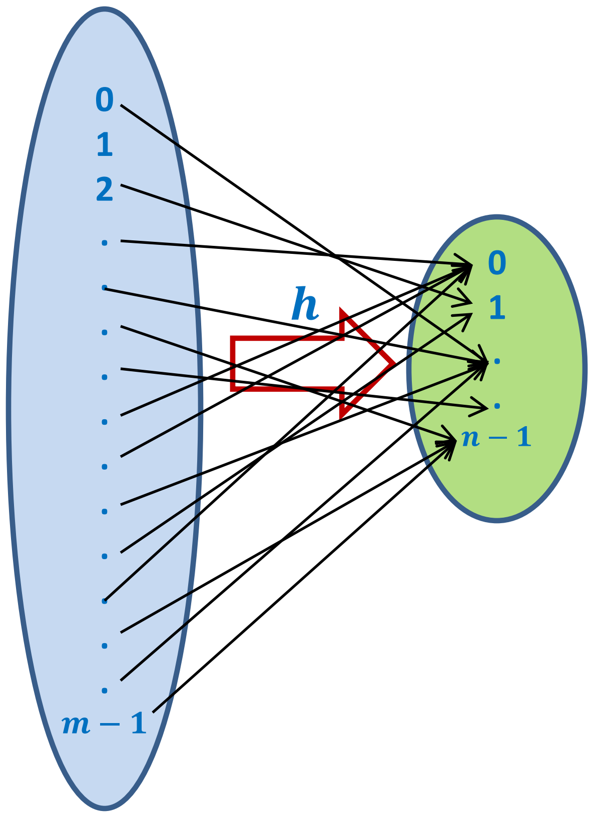
Efficiency of Hashing depends upon hash function

A hash function h is good 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 **every** S.

Hash function, hash value, hash table



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

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

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

➔ **O**($\boldsymbol{n}$) worst case search time.

Hashing: Practice

Designed in 1953 by as a heuristic

Practice:

- **The function $h(i)$ = i mod n** works **very well**
- **Hashing** is preferred to BST most of the times.

1953

Reason: S is usually a **uniformly random** subset of U.

➔ Average search time is **O**(1).

Question: Can we achieve worst case **O**(1) search time using hashing ?

Yes

[FKS] Fredman, Komlos, Szemeredy, *Journal of ACM*, volume 31, **1984**

1984

Though no hash function is good for **every** S.
There are quite large number of hash function which will be good for any **given** fixed S. [**FKS**] find such hash functions in an elegant manner.

Hashing: theory

U : $\{0,1, \dots, m-1\}$
$S \subseteq U$,
$n = |S|$,

Theorem [FKS]: A hash table and hash function can be computed in **O**(n) **time** for a **given** S s.t.
Space : **O**(n)
Query time: worst case **O**(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
Lecture 27

	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)**

No one even tried to find out ☹

What makes Quick sort popular ?

No. of repetitions = $\mathbf{1000}$

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

Inference:

The chances of deviation from average case decreases as $\boldsymbol{n}$ increases.

➔ The *reliability* of quick sort increases as $\boldsymbol{n}$ increases.

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 we 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**.

Miscellaneous problems

Problem 1

Input:

Given an array **A** storing n numbers,

there is an $i < n$ (unknown) s.t.

A$[0] <$ **A**$[1] <$ …$<$ **A**$[i] >$ **A**$[i+1] >$… $>$**A**$[n-1]$

Aim:

To search efficiently

Answer : **O**(log n) is possible

Problem 2

Input:

Given an array **A** storing n numbers,

there is an $i < n$ (unknown) s.t.

A$[0] \leq$ **A**$[1] \leq \ldots \leq$ **A**$[i] \geq$ **A**$[i+1] \geq \ldots \geq$ **A**$[n-1]$

Aim:

To search efficiently

Answer : No algorithm can search **A in better** than **O**(n) time in worst case.

Problem 3

Input:

Given an array **A** storing n numbers,

there are $i < j < n$ (unknown) s.t.

A$[0] <$ **A**$[1] < \ldots <$ **A**$[i] >$ **A**$[i+1] > \ldots >$ **A**$[j] <$ **A**$[j+1] < \ldots <$ **A**$[n-1]$

Aim:

To search efficiently

Answer : No algorithm can search **A in better** than **O**(n) time in worst case.

We shall find inspiring answers to these problems and many more in the last class of this course. So don't miss it ☺ …