

# Today - Lecture 7 - CS163

\* Begin Topic #6 on "Table ADT's"

\* We will discuss:

- what is a Table ADT

- how efficient are the data structures

- learn a new data structure

(hash table with chaining — Prog<sup>#3</sup>)  
    ↑ an array of LLL

## Announcements:

# Table Abstractions

- based on the **VALUE** of the data  
(not the position)

	Add	Remove	Search	Display
Sorted Array	+ Binary Search - Shifting - Memory	+ Binary Search - Shifting	+ <b>Binary Search</b>	+ <b>Displays in Sorted Order automatically</b>
Unsorted Array	+ <b>Direct Access</b> + <b>No Shifting</b> - Memory	- Sequential Search - Shifting	- Sequential Search	- Must implement a sorting algorithm
Sorted LLL	- Sequential Search + No Shifting + Flexibility with Memory	- Sequential Search + No Shifting + Can Stop early if there is no match	- Sequential Search	+ <b>Supported</b>
Unsorted LLL	+ <b>Direct Access</b> + <b>No Shifting</b> + <b>Flexibility with Memory</b>	- Sequential Search + No Shifting + Flexibility with Memory	- Sequential Search	- Must implement a sorting algorithm

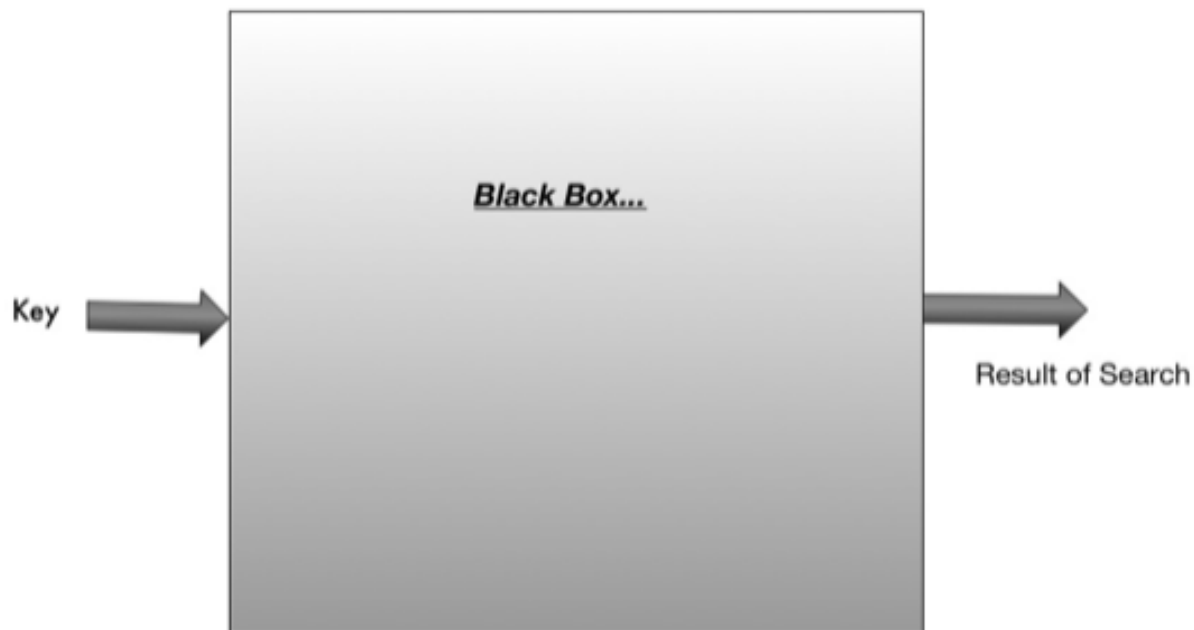
With hash tables

\* using a "collision resolution" technique  
of chaining

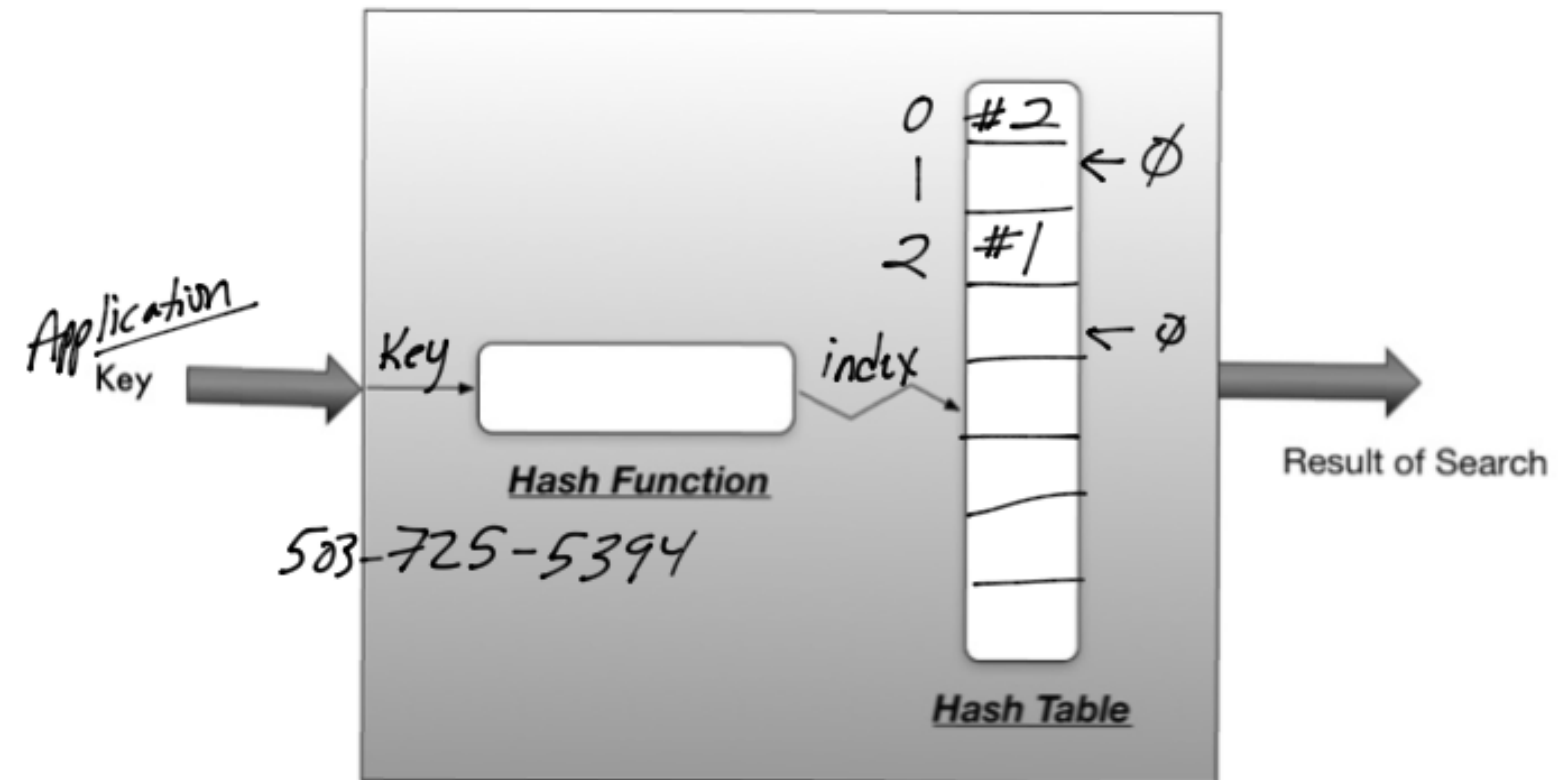
	Add	Remove	Search	Display
Sorted Array	+ Binary Search - Shifting - Memory	+ Binary Search - Shifting	+ <b>Binary Search</b>	+ <b>Displays in Sorted Order automatically</b>
Sorted LLL	- Sequential Search + No Shifting + Flexibility with Memory	- Sequential Search + No Shifting + Can Stop early if there is no match	- Sequential Search	+ <b>Supported</b>
Hash Table using Chaining	+ <b>Instantaneous</b> + <b>Direct Access</b> + <b>Flexibility with Memory</b>	+ <b>Instantaneous</b> + <b>Direct Access</b> + <b>Flexibility with Memory</b>	+ <b>Instantaneous</b> + <b>Direct Access</b> + <b>Flexibility with Memory</b>	- Not Available (would need to use an alternate data structure)

## Hash Tables from a "client" perspective

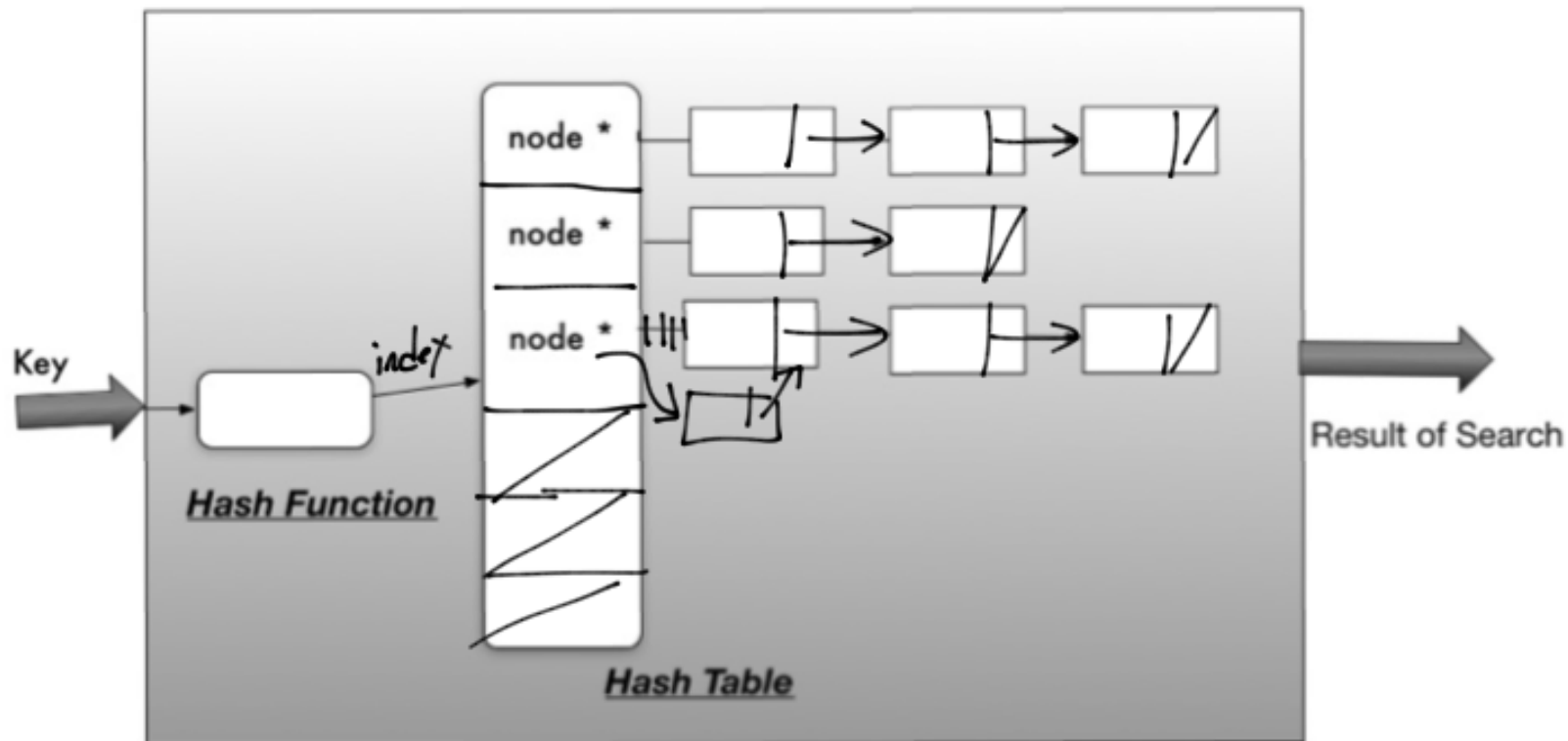
\* Remember, the client is only aware that they are interested in finding information based on the "value" of the search request!



# Hash Table using Linear Probing



Hash tables with chaining (as the collision resolution technique)



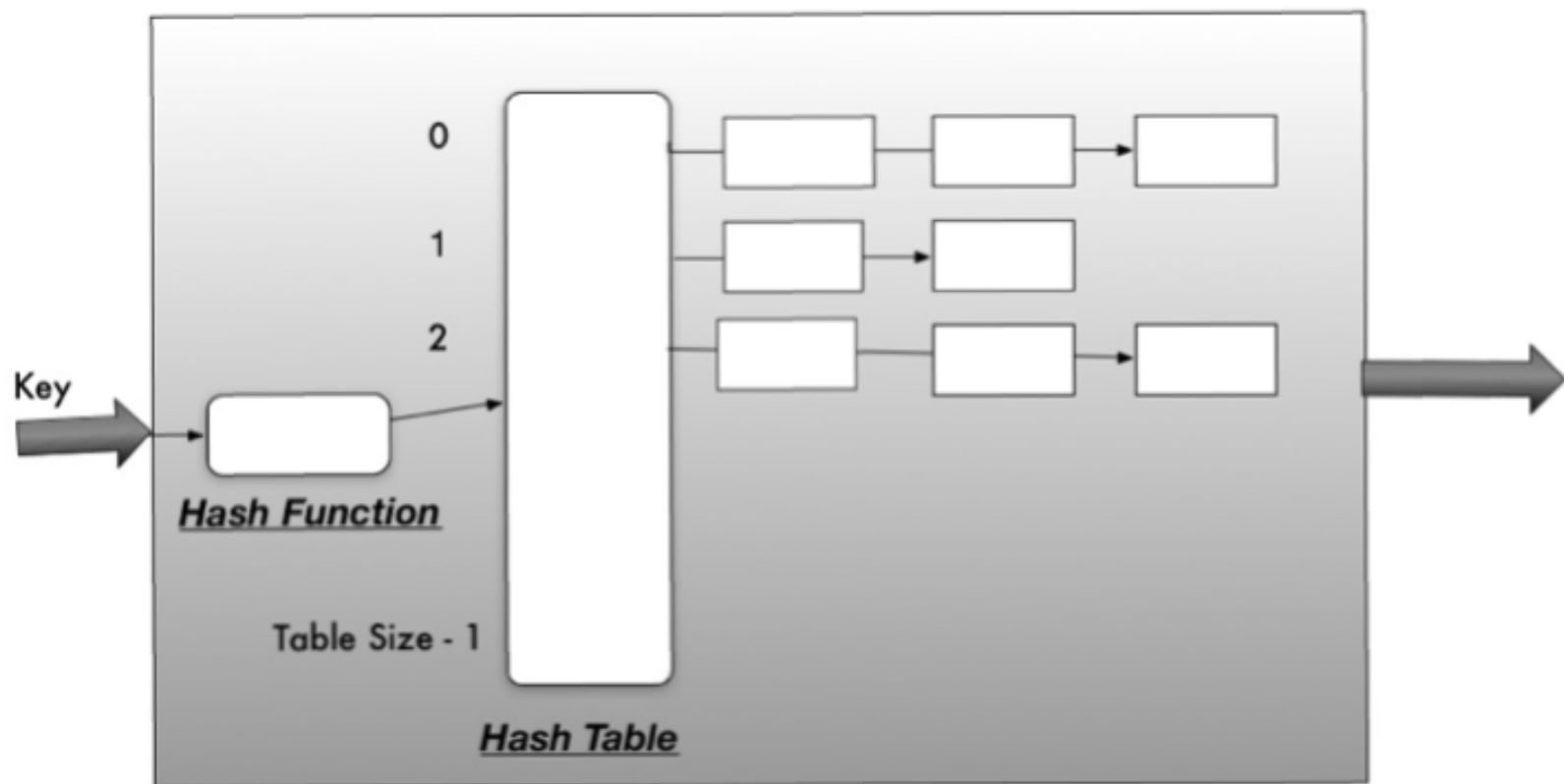
(Add will insert at the head of a chain even if there is a collision.  
chains are not ordered)

# "Chaining"

- 1) collision resolution technique
- 2) hash table is an array of head ptrs
- 3) The array must be initialized so that each "head" pointer is NULL
- 4) We will know that there is a collision because "head" (for that array element) is NULL
- 5) chains are not sorted
- 6) The length of a chain represents the number of collisions

# Hash Functions and Hash Table Size

- 1) Goal - distribute data across chains
- 2) Hash table size should be a prime #
- 3) Examine "Key Value" to determine how to write a hash function that will map all or part to an index ( $\phi \rightarrow \text{TableSize} - 1$ )

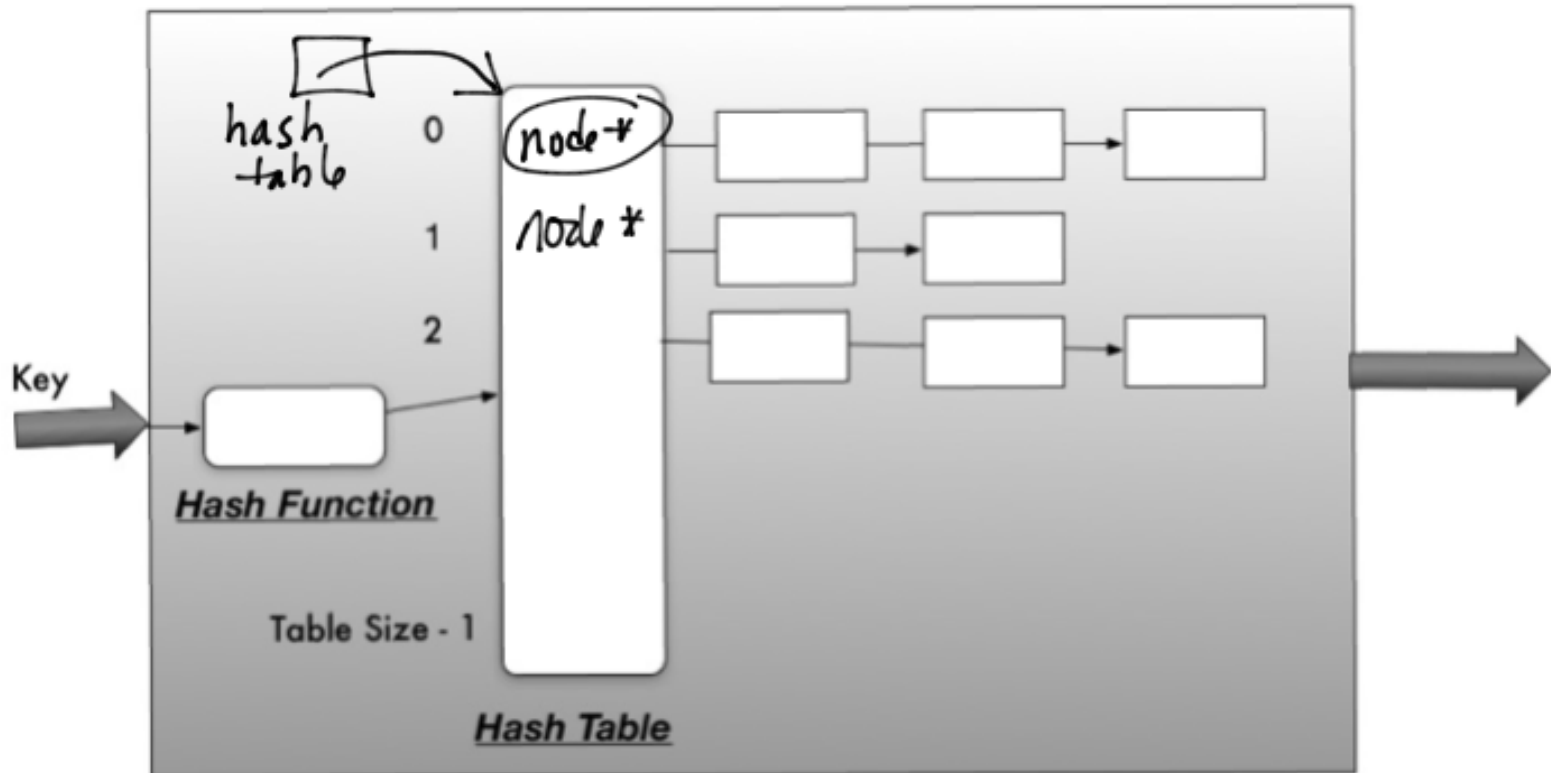




More ideas:

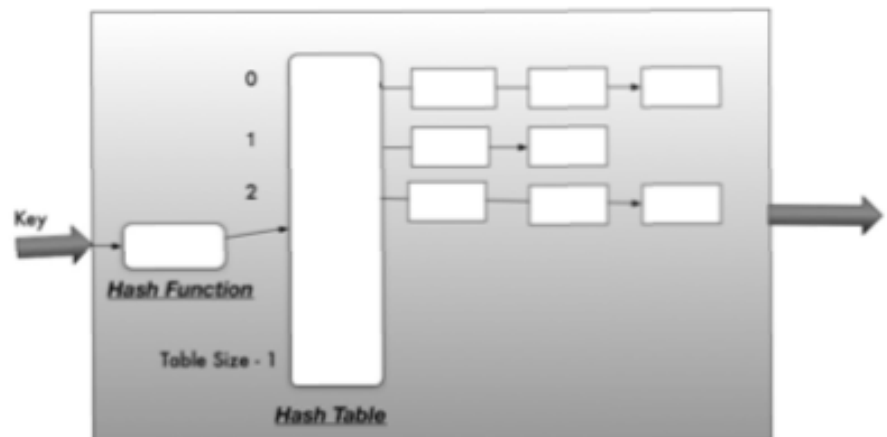
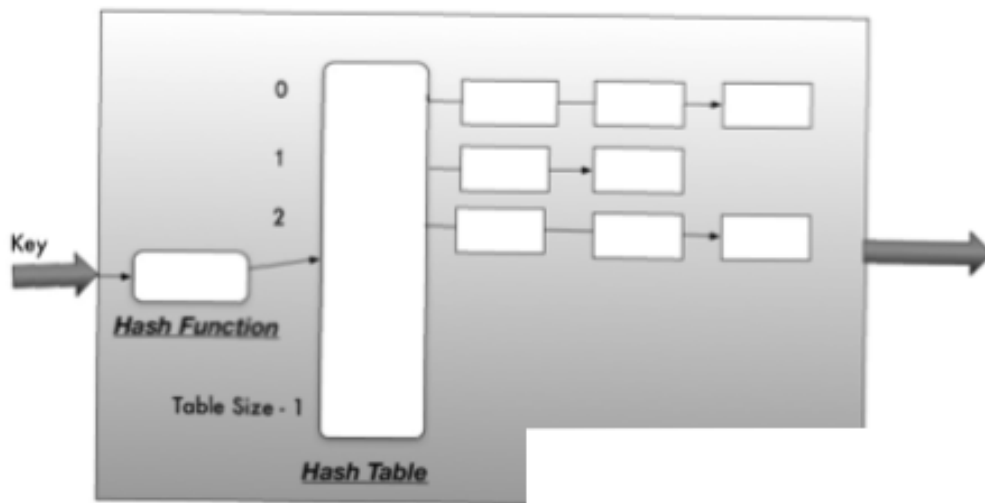
$$\text{index} = (\text{name}[\phi] + \text{name}[2]) \% \text{size};$$

why not [1]?



# Working with multiple Key Values : (multiple hash tables)

Array or Other data structure that contains the Actual Data (possibly maintained in sorted order by one of the key values).



# Problems

1. Key  $\rightarrow$  Index  $\begin{cases} \rightarrow \text{perfect function} \\ \rightarrow \text{many collisions} \end{cases}$

2. NOT Sorted  $\rightarrow$  Requires a different data struct.

3. NO Range checking— MUST have EXACT Match!