CSC 212
Data Structures & Algorithms

Fall 2022 | Jonathan Schrader

Hash Tables

Housekeeping

Lab 11: Sets and Maps

Assignment 5

- Due 12/2 11:59p

Term Project

- Due 12/5 11:59p

HASHING

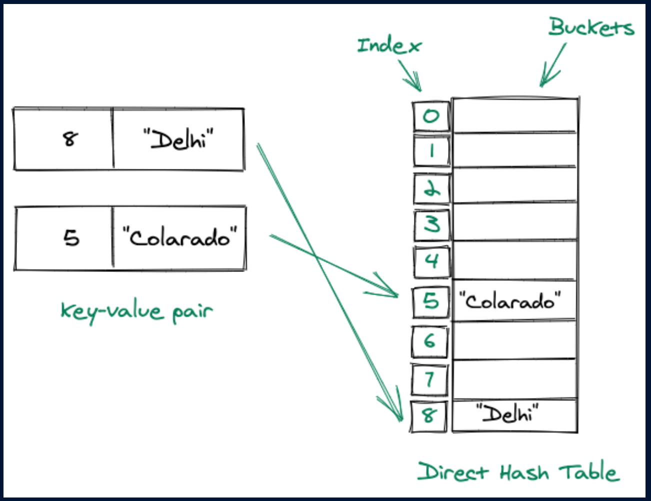
Storing data

implementation	guarantee			average case			ordered ops?	key interface
	search	insert	delete	search	insert	delete		
sequential search (unordered list)	n	n	n	n	n	n		`equals()`
binary search (ordered array)	$\log n$	n	n	$\log n$	n	n	✔	`compareTo()`
BST	n	n	n	$\log n$	$\log n$	$\sqrt{n}$	✔	`compareTo()`
red-black BST	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$	✔	`compareTo()`
hashing	n	n	n	$1^{\dagger}$	$1^{\dagger}$	$1^{\dagger}$		`equals()` `hashCode()`

Summary Table

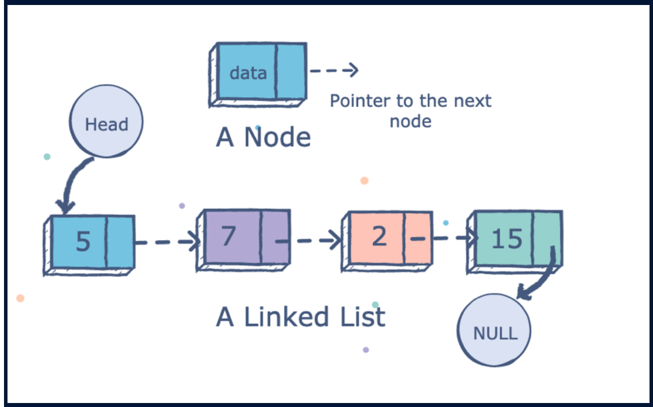
Hash Tables

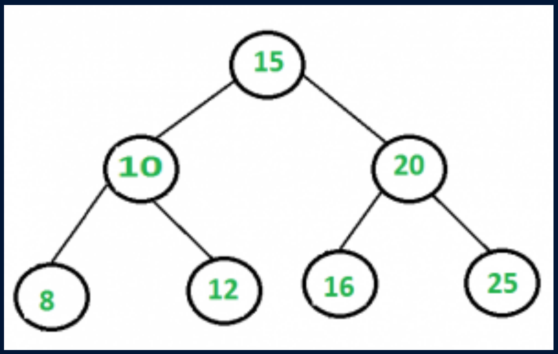
- implements an associative array or dictionary
- an abstract data type that maps keys to values
- uses a hash function to compute an $index$, also called a $hashcode$
- at lookup, the key is hashed and the resulting hash indicates where the corresponding value is stored.

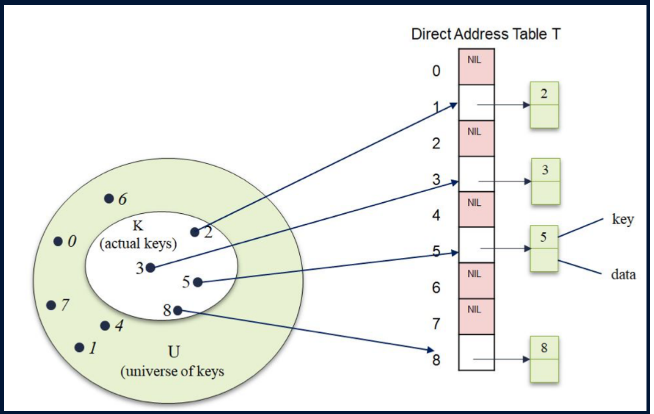


Hash Table

Why not…







Arrays & Linked Lists

- Search $O(log\ n)$
- Insert/Delete, much more costly

Balanced BST

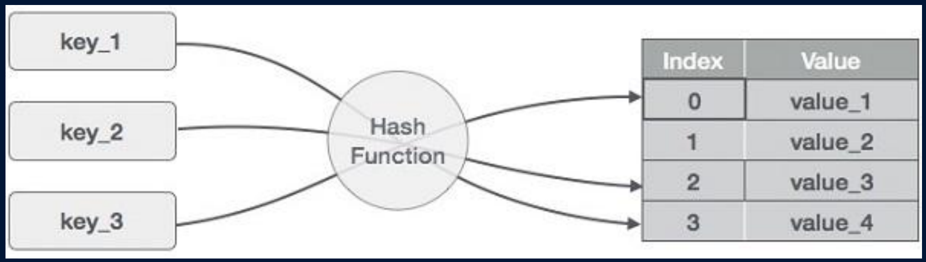
- Guarenteed $O(log\ n)$

Direct Access Table

- Best-case $O(1)$
- Practical limitations
  - Extra space
  - A given integer in a programmming language may not store n digits
  - Therefore, not always a viable option

Hash Functions

- a function converting a piece of data into a smaller, more practical integer
- the integer value is used as the $index$ between 0 and $m-1$ for the data in the hash table
- ideally, maps all keys to a unique slot $index$ in the table
- perfect hash functions may be difficult, but not impossible to create



Properties of good hash functions:

- Efficiently computable
- Should uniformly distribute the keys (each table position equally likely for each)
- Should minimize collisions
- Should have a low load factor $\frac{\#\ items\ in\ table}{table\ size}$

Modular Hashing

To uniformly create hashes, hash functions may use heuristic techniques of division or multiplication

Legend

h = hash function
x = key
HT = hash table

m = table size
b = buckets
r = items per bucket

Rule

$$0 \leq h(x) < m$$
$$or,$$
$$0 \leq h(x) < b - 1$$

$$Entry\ lookup \Rightarrow HT[h(x)]$$

Syntax

$$\begin{aligned} h(x) &= x \ mod \ m \\ &= x \ \% \ m \end{aligned}$$

Example

Suppose there are six students a1, a2, a3, a4, a5, a6 in the Data Structures class and their IDs are a1: 197354863; a2: 933185952; a3: 132489973; a4: 134152056; a5: 216500306; and a6: 106500306.

$$h : \{k_1, k_2, k_3, k_4, k_5, k_6\} \rightarrow \{0, 1, 2, \ldots 12\} \ by \ h(k_1) = k_1 \% \ 13$$

$$\begin{aligned} h(k_1) &= 197354863 \ \% \ 13 = 4 & h(k_4) &= 134152056 \ \% \ 13 = 12 \\ h(k_2) &= 933185952 \ \% \ 13 = 10 & h(k_5) &= 216500306 \ \% \ 13 = 9 \\ h(k_3) &= 132489973 \ \% \ 13 = 5 & h(k_6) &= 106500306 \ \% \ 13 = 3 \end{aligned}$$

Suppose $HT[b] \leftarrow a \ldots$

$$\begin{array}{lll} HT[4] \leftarrow 197354863 & HT[5] \leftarrow 132489973 & HT[9] \leftarrow 216500306 \\ HT[10] \leftarrow 933185952 & HT[12] \leftarrow 134152056 & HT[3] \leftarrow 106500306 \end{array}$$

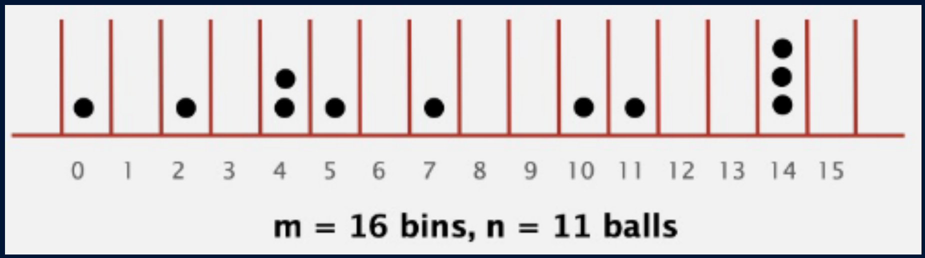
Uniform Hashing

Assumption

- Any key is equally likely (*and independent of other keys*) to hash to one of m possible indices

Bins and Balls

- Toss n balls uniformly at random into m bins



Bad News [birthday problem]

- In a random group of 23 people, more likely than not that two people share the same birthday
- Expect two balls in the same bin after $\sim \sqrt{\pi m/2} \qquad // = 23.9\ when\ m = 365$

Good News

- when $n >> m$, expect most bins to have $\approx \frac{n}{m}$ balls
- when $n = m$, expect most loaded bin has $\sim \frac{ln\ n}{ln\ ln\ n}$ balls

COLLISIONS

collisions

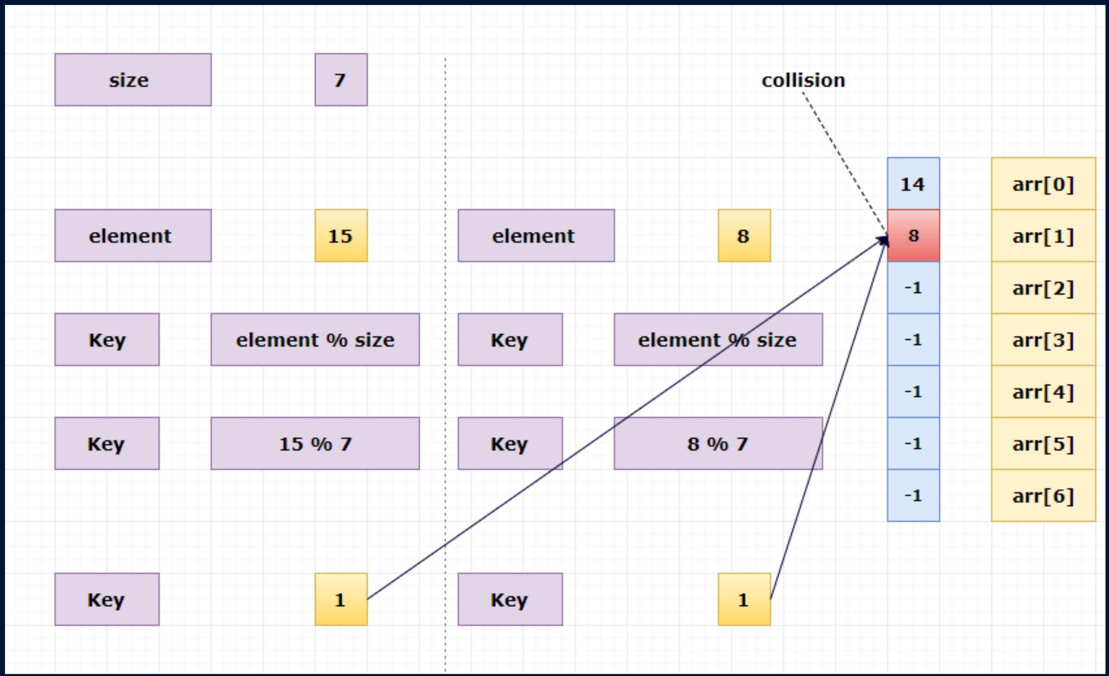
Two distinct keys that hash to the same index

birthday problem

$\Rightarrow$ can't avoid collisions

load balancing

$\Rightarrow$ no index gets too many collisions
$\Rightarrow$ ok to scan though all colliding keys



Collision

SEPARATE CHAINING

Simple Uniform Hashing

- keeps a list of all elements that hash to the same value

Performance

m = Number of slots in hash table
n = Number of keys to be inserted in hash table

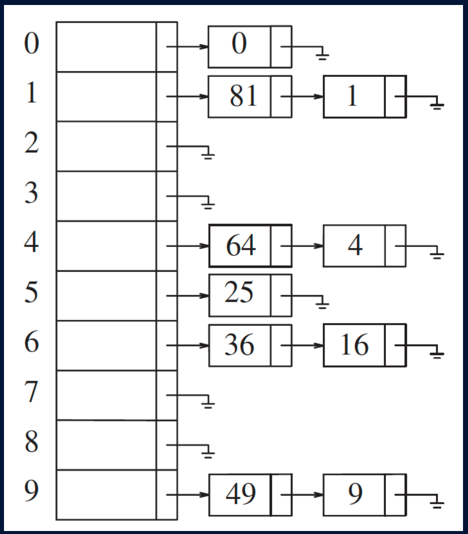
Load factor $\alpha = n/m$
Expected time to search = $O(1+\alpha)$
Expected time to delete = $O(1+\alpha)$

Time to insert = $O(1)$
Time complexity of search, insert, and delete is $O(1)\ if\ \alpha\ is\ O(1)$

Example

$$h : 0, 81, 64, 25, 36, 49, 1, 4, 16, 9$$

$$h(k_1) = 0 \ \% \ 10 = 0 \quad \Rightarrow \quad HT[0] \leftarrow 0$$
$$h(k_2) = 81 \ \% \ 10 = 1 \quad \Rightarrow \quad HT[1] \leftarrow 81$$
$$h(k_3) = 64 \ \% \ 10 = 4 \quad \Rightarrow \quad HT[4] \leftarrow 64$$
$$h(k_4) = 25 \ \% \ 10 = 5 \quad \Rightarrow \quad HT[5] \leftarrow 25$$
$$h(k_5) = 36 \ \% \ 10 = 6 \quad \Rightarrow \quad HT[6] \leftarrow 36$$
$$h(k_6) = 49 \ \% \ 10 = 9 \quad \Rightarrow \quad HT[9] \leftarrow 49$$
$$h(k_7) = 1 \ \% \ 10 = 1 \quad \Rightarrow \quad HT[1] \leftarrow 1$$
$$h(k_8) = 4 \ \% \ 10 = 4 \quad \Rightarrow \quad HT[4] \leftarrow 4$$
$$h(k_9) = 16 \ \% \ 10 = 6 \quad \Rightarrow \quad HT[6] \leftarrow 16$$
$$h(k_{10}) = 9 \ \% \ 10 = 9 \quad \Rightarrow \quad HT[9] \leftarrow 9$$



A separate chaining hash table

OPEN ADDRESSING

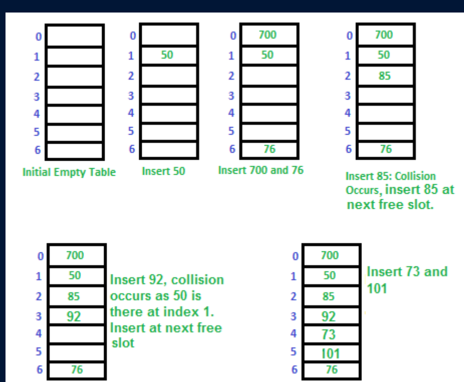
Linear Probing

- keeps a list of all elements that hash to the same value

Rule

$h_i(x) = (Hash(x) + i)$

If $h_0(x) = (Hash(x) + 0)$
If $h_1(x) = (Hash(x) + 1)$
If $h_2(x) = (Hash(x) + 2)$
… and so on



Example

$h : \{50, 700, 76, 85, 92, 73, 101\}$

$h_0(50) = 50 \quad \% \, 7 = 1$

$h_0(700) = 700 \ \% \, 7 = 0$

$h_0(76) = 76 \ \% \, 7 = 6$

$$h_0(85) = 85 \ \% \, 7 = 1$$
$$\Rightarrow h_1(85) = (85 + 1) \ \% \, 7 = 2$$

$$h_0(92) = 92 \ \% \, 7 = 1$$
$$\Rightarrow h_1(92) = (92 + 1) \ \% \, 7 = 2$$
$$\Rightarrow h_2(92) = (92 + 2) \ \% \, 7 = 3$$

$$h_0(62) = 62 \ \% \, 11 = 7\$$$
$$\Rightarrow h_1(62) = (62 + 1) \ \% \, 11 = 8\$$$
$$\Rightarrow h_2(62) = (62 + 2) \ \% \, 11 = 9\$$$

$h_0(73) = 73 \ \% \, 7 = 4$

$h_0(101) = 101 \ \% \, 7 = 5$

Quadratic Probing

Rule

$$h_i(x) = (Hash(x) + i^2)\ \%\ \ HashTableSize$$
$$\Rightarrow (Hash(x) + i * i)\ \%\ \ HashTableSize$$

$$If\$h_0(x) = (Hash(x) + 0^0)\ \%\ \ HashTableSize\$$$
$$If\$h_1(x) = (Hash(x) + 1^1)\ \%\ \ HashTableSize\$$$
$$If\$h_2(x) = (Hash(x) + 2^2)\ \%\ \ HashTableSize\$$$
$$\ldots and\ so\ on\ if\ h_i\ is\ already\ full\ldots$$

0
1
2
3
4
5
6

Example

h : { 50 700 76 85 92 73 101 }

$$h_0(50) = 50\quad \%\ 7 = 1$$
$$h_0(700) = 700\ \ \%\ 7 = 0$$
$$h_0(76) = 76\ \ \%\ 7 = 6$$
$$h_0(85) = 85\ \ \%\ 7 = 1$$
$$\Rightarrow h_1(85) = 85 + (1 * 1)\ \ \%\ 7 = 2$$
$$h_0(92)f = 92\ \ \%\ 7 = 1$$
$$\Rightarrow h_1(92) = 92 + (1 * 1)\ \ \%\ 7 = 2$$
$$\Rightarrow h_2(92) = 92 + (2 * 2)\ \ \%\ 7 = 5$$
$$h_0(73) = 73\ \ \%\ 7 = 3$$
$$\Rightarrow h_0(73) = 73 + (1 * 1)\ \ \%\ 7 = 4$$
$$h_0(101) = 101\ \ \%\ 7 = 3$$

Double Hashing

Rule

$$H1(x) = Hash1(x) \ \% \ \ HashTableSize$$
$$H2(x) = Hash2(x) \ \% \ \ HashTableSize$$
$$\Rightarrow //random \ mod \ function$$
$$\Rightarrow 1 + (x \ \% \ 5)$$

$If \ h_0(x) = (Hash(x)\%HashTableSize$
$If \ h_{i+1}(x) = (Hash(x) + 1 * (Hash2(x))\%HashTableSize$
… and so on

$$h_0(73) = 73 \ \ \% \ 7 = 3$$
$$h_1(73) = (73 + (1 + (73 \ \% \ 5)) \ \ \% \ 7 = 4$$

$$h_0(101) = 101 \ \ \% \ 7 = 3$$
$$h_1(101) = (101 + (1 + (101 \ \% \ 5)) \ \ \% \ 7 = 4$$

0
1
2
3
4
5
6

Example

h : { 50 700 76 85 92 73 101 }

$$h_0(50) = 50 \quad \% \ 7 = 1$$

$$h_0(700) = 700 \ \ \% \ 7 = 0$$

$$h_0(76) = 76 \ \ \% \ 7 = 6$$

$$h_0(85) = 85 \ \ \% \ 7 = 1$$
$$h_1(85) = (85 + (1 + (85 \ \% \ 5)) \ \ \% \ 7 = 2$$

$$h_0(92) = 92 \ \ \% \ 7 = 1$$
$$h_1(92) = (92 + (1 + (92 \ \% \ 5)) \ \ \% \ 7 = 4$$

$$h_0(73) = 73 \ \ \% \ 7 = 3$$

$$h_0(101) = 101 \ \ \% \ 7 = 3$$
$$h_1(101) = (101 + (1 + (101 \ \% \ 5)) \ \ \% \ 7 = 5$$

Comparison

Linear Probing

- Easy to implement
- Best cache performance
- Suffers from clustering

Quadratic Probing

- Average cache performance
- Suffers less from clustering

Double Hashing

- Poor cache performance
- No clustering
- Requires more computation time