Week 6

Assignments:
Program 2: is being graded
Program 3: available soon and due before 10pm on Thursday 3/14
Homework 5: available soon and due before 10pm on Monday 3/4
X-Team Exercise #2: due before 10pm on Thursday 2/28
X-Team Exercise #3: due before 10pm on Monday 3/4

MIDTERM EXAM
THURSDAY MARCH 7th, 5:00 pm – 7:00 pm

- **Lecture 001 students: Room 6210 of Social Sciences Building**
- **Lecture 002 students: Room B10 of Ingraham Hall Building**
- **Lecture 004 students: Room 125 of Agriculture Building**

- **Bring**
  - **UW ID**
    Note: if you do not have your UW ID, you will be asked to wait until after students with ID have been admitted
  - **#2 Pencils**
  - **good eraser**
- **At Exam:**
  - **arrive early**
  - **get ID scanned**
  - **get scantron form**
  - **find seat directly behind another student**
- **See posted exam information**

Read: Module 6 for this week and 7 for next week

THIS WEEK:

- Hashing
- Ideal Hashing
- Techniques for generating hash codes
- Handling `String` keys
- Handling `double` keys
- Choosing Table Size
- Resizing a hash table
- Collision Handling

NEXT WEEK:

- Graphs
- Exam Review
- Midterm Exam

Hashing

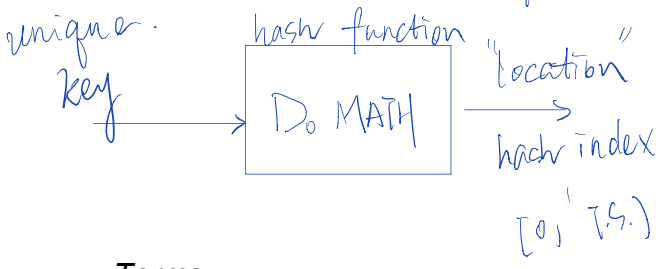
Goal: Do better than $O(\log_2 N)$

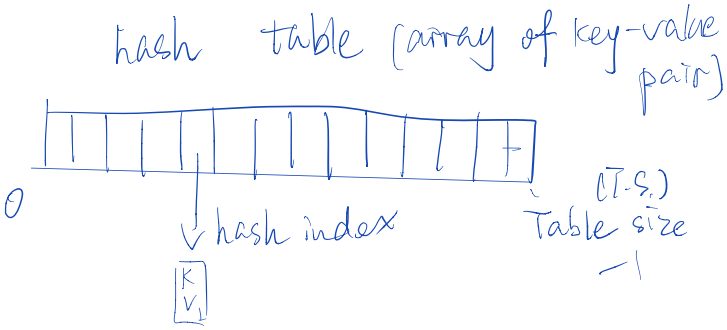
look up, insert, delete $O(1)$

Concept: "know where it is" = compute where it is.

(MATH).

create a function





Terms:

key unique info for each (K,V) pair

hash function convert key to a hash index

hash index item's location (index) in hash table.

hash table array that stores references to (K,V) pairs.

table size (TS) length, size, capacity of hash table (array).

load factor (LF) # items / Table size. (how full is the table). 1 = # item = T.S.
0.5 = half full.

P3 → threshaded for resizing

Java's **hashCode()** method

-2,147,483,648 ~ Integer value for an object - Intermediate value may be bigger then array (smaller).

Ideal Hashing

Assume
- need to store 150 students records
- table is an array of student records
- null is sentinel value meaning that table element is unused
- key is the student's id number, and it is one of the following 5 digit integers

  11000, 11001, 11002, … 11048, 11049, … 11148, 11149

index 0 1 2 ... 149.

→ What would be a good hash function to use on the ID number?

```
int hash(K key) {
    return key - 11000;
}
```

perfect hash function

Trivial Hash Function: if key is an int, use it - put in correct range.

Perfect Hash Function: each key match an unique hash index. no collisions.

If Perfect Hash,

```
void insert(K key, D data) { table[hash(key)] = data ; }

D    lookup(K key)         { return table[hash(key)]; }

void delete(K key)         { table[has(key)] = null ; }
```

The UW uses 10 digit ID numbers: 9012345789 9012345432 9023456789

→ Is a perfect hash function possible for these id numbers?

perfect function
but waste spaces.

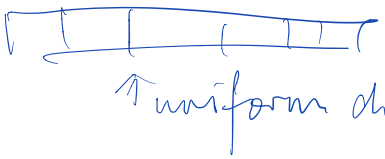
→ Would the last 3 digits of the ID work as above?

Collision: when different keys map to same hash index.

Key Issues:
- design a "good" hash function
- choose a good table size.
- handle collisions.

Designing a Hash Function

Good Hash Functions:

1. must be deterministic.
2. should achieve uniforming ⇒ 
   ↑ uniform distribution
3. should minimize collisions
4. fast / easy to compute for a computer.

Java's hashCode function:

What is it? built in method in object class

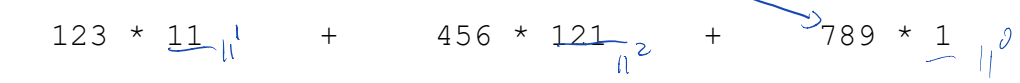
How can we use it?

1. get the hashCode for the "key" ← combined fields.
2. convert to a valid hash index [0, TS)

```
Integer i = new Integer(-1);
hashIndex = Math.abs(i.hashCode()) % TS;
```

Techniques for Generating Hash Codes

Integer Key 90123456789



Extraction – divide into parts
– use distinct parts

Weighting – emphasize some parts over others.
$*11^1$ $*11^2$ $*11^0$

Folding – combine back together into a int
– add or exclusive-OR

123×11
$+ 456 \times 11^2$
$+ 789 \times 11$
$=$ hashCode

hashIndex = Math.abs(i.hashCode()) % TS.length;

Handling `String` Keys

C_i the ASCII code for character at pos i in a string.

CAT

C_0 C_1 C_2

Java: $C_0 * 31^2 + C_1 * 31^1 + C_1 * 31^0$

$= 67 \times 31^2 + 65 \times 31 + 84 * 1$

"CAT". hashCode = 66486

why 31?

- multiply, divide by 2 it is easy for computer

v = 3 0011

v * 2 = 6 0110 left shift by 1 = multiply by 2.

v << 1 0110

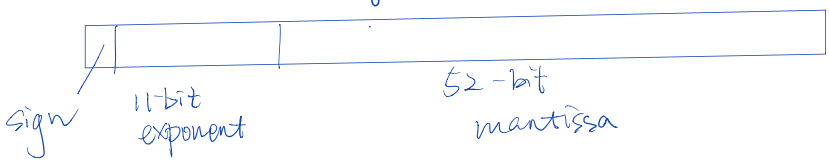
v * 4 1100 <=> v << 2

(v * 32) − v = v * 31

=> (v << 5) − v

Handling `Double` Keys

64-bit IEEE Floating Point Real number



3.678×10^{-12}

0X 4 1 6 1 B5 14 40

Java hashCode.

int = upper + lower

eg. v = 0110 1011

v >> 4 = 0110

exclusive-OR 1011 => hashCode

```
long bits = Double.doubleToLongBits(number);
int hashCode = (int)(bits ^ (bits >> 32));
```

shift upper to lower.

XOR operator.

cast to int

lower 32

upper

Choosing the Table Size

Table Size and Collisions

Assume 100 items with random keys in the range 0 – 9999 are being stored in a hash table.
Also, assume the hash function is: (key % table_size).

→ How likely would a collision occur if the table had room for 10000 elements? 1000? 100?

ITEMS	*T.S.*	*Expected # of collisions*	*L.F.* (#items / TS)
100	10,000	0 or 1	0.01
100	1,000	3 ~ 7	0.10
100	100	35 ~ 47	1.00
100	10		10.00

as load factor ↑ collison ↑

↓ choose table size with extra space

↓ ? How much extra space ?

* to keep efficiency, minimize collisions.

Table Size and Distribution

Now, assume 50 items are stored in a hash table.
Also assume the hashCode function returns multiples of some value x.
For example, if x = 20 then hashCode returns 20, 40, 60, 80, 100, ...

accidental not intentional.

How likely would a collision occur if the table had 60 elements? 50? 37?

N	*TS*	*# collisions*
50	60	47
50	50	45
50	37	13

→ 3 elements used
→ 5
→ 37.

TS / G.C.F(X, TS) = element used.

60 / G.C.F(20, 60) = 3 elements

So / G.C.F(20, 50) = 5.

37 / G.C.F(20, 37) = 37.

prime number.

Resizing the Hash Table

* Resize when its "full"? ← reached load factor threshold

Naïve Expand 1. make new big table
2. copy from old to new

30		17	88	

TS = 5

Key % TS.

30		17	88						

TS = 10

↑ not work.

Rehashing

1. Double TS to nearest prime — O(1)
must rehash each item into new double — O(N)
reassign ht variable to refer to new table

2.

0						17		30		

11

Complexity $O(N) = O(1) + O(N)$
↑ resize, ↑ rehash

Collision Handling using Open Addressing

Open Addressing

* each element stores one item
* if collison, look for an open address.

Probe Sequence: finding an open address and matching key

Linear Probing step by step until open address

must wrap around table

P.S. H_K ← hash index of key k.

$H_K \rightarrow H_{K+1} \rightarrow H_{K+2}$ ----- (keep looking for next key if it's full).

166 % 15 = 166 % 11 = 1

359 359 % 11 = 7 - full → 8

263 263 % 11 = 10 → 0 → 1 → 2 → 3 → 4 → 5

0	1	2	3	4	5	6	7	8	9	10
440	166	266	124	246	263		337	359		351

con: "primary clusting"

pro: can find an open address, if one exists

⇒ Find: get index, wrap around, find ✓

open index → not in

⇒ remove / delete 263. – mark the address "REMOVED" → not be

Collision Handling using Open Addressing

Quadratic Probing

P.S. H_k, $H_k + 1^2$, $H_k + 2^2$ ..... wrap around the table.

pro: avoid primary clustering

con: may not find an open address.

"can lead to secondary clustery"

166 % 11 = 1

359 = 7, 8

263 = 10, 0, 3, 8, 4, 2, 2, 4, 8, 3, 0, 10, 0, 3 .....

440	166	266	124	246			337	359		351
								8		10

Double Hashing – define a second hash function to compute step size

P.S H_k, $H_k + 1 * \underbrace{hash2(key)}_{\text{step-size}}$, $H_k + 2 \times SS$, ... (wrap around).

probe sequences assuming H_k is index 0:

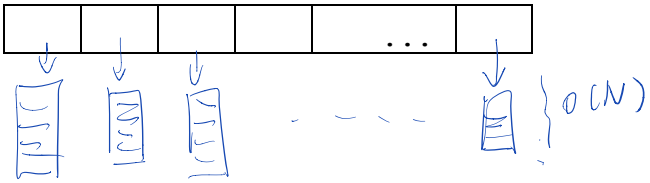
Step size	Table size 10	Table size 11
2	0, 2, 4, 6, 8, 0, ...	0, 2, 4, 6, 8, 10, 1, 3, 5, 7, 9
5	0, 5, 0	0, 5, 10, 4, 9, 3, 8, 2, 7, 1, 6

Collision Handling using Buckets

Buckets each element stores a bucket, that can store multiple items.

Idea: if collison, add to array bucket.

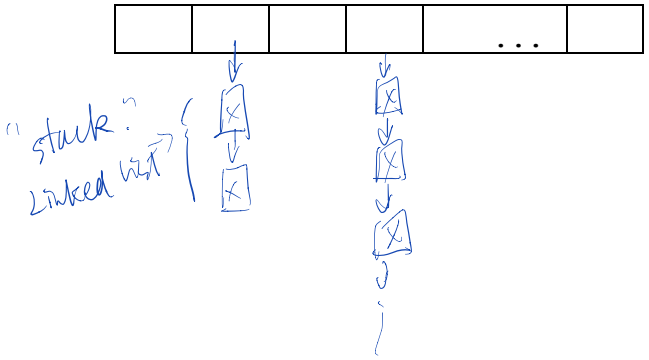
Array Buckets - fixed size



← need extra space
what happens when any bucket is full.

+ easy to implement
+ no wasted space.
- long chains. maybe, O(N)

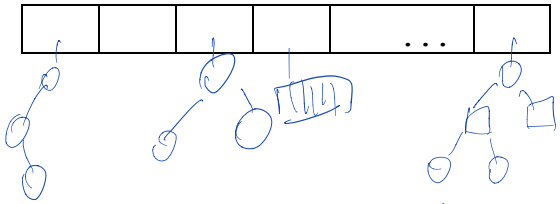
"Chained" Buckets



+ easy to implement
does not over fill
— — waste space

- long chains can lead to poor lookup.
+ buckets can grow / shrink.

Tree Buckets BST, AVL, RBT.



can be any balanced search tree.
guards against worst case O(N)
O(log N).

↓ hash table bucket.

Java API Support for Hashing

`hashCode` method

- method of Object class
- returns an int
- default hash code is BAD - computed from object's memory address

Guidelines for overriding `hashCode`:

`Hashtable<K,V>` and `HashMap<K,V>` class

- in java.util package
- implement `Map<K,V>` interface
  `K` type parameter for the key
  `V` type parameter for the value

  operations:

- constructors allow you to set
  initial capacity (default = 16 for HashMap, 11 for HashTable)
  load factor (default = 0.75)
- handles collisions with chained buckets
- HashMap only:
- Hashtable only:

TreeMap vs HashMap

	TreeMap	HashMap
Underlying d.s.	red-black tree	hash table
Complexity of basic ops. insert lookup remove	B.C O(1) log(2N)	B.C, A.C, W.C O(1) O(1) O(N) O(1) O(1) O(N) O(1) O(1) O(N)
Iterating over keys	pre - order in - ascending post level sorted	no sorted order
Complexity of iterating over values	O(N) where N is # of key	O(TS) O(N) ⇒ O(TS+N)