# CS 106 INTRODUCTION TO DATA STRUCTURES

SPRING 2020

PROF. SARA MATHIESON

HAVERFORD COLLEGE

# ADMIN

- Lab 6 due Tuesday
- Welcome prospective students!
- May need to do random breakout rooms since many people are not signed into Zoom
- Lab tomorrow: same plan (will start at 9am so feel free to come early)
  - Make sure to sign-in!

*Remind me to RECORD*

# REVISED TA/OFFICE HOURS

Sunday 7-9pm (Juvia)

Monday 8-midnight (Steve)

Tuesday 11:30-12:30pm (Lizzie)

Tuesday 4:30-6pm (Sara)

Wednesday 8-midnight (Steve)

**Thursday 11:30-12:30pm (Lizzie)**

**Thursday 9-11pm (Will)**

**Friday 8-10pm (Gareth)**

*Today/Tomorrow*

Saturday 4-6pm (Will)

Saturday 8-10pm (Gareth)

# LAB 6 OUTPUT

Call removeMax on each element in the heap to remove it and print it out

Break ties based on last name (still A-Z, which will look like Z-A in the printout)

```
Joseph R. Biden Jr.:27.0
Bernard Sanders:20.0
Elizabeth Warren:18.0
Kamala D. Harris:12.0
Pete Buttigieg:7.0
Beto O'Rourke:2.0
Tulsi Gabbard:2.0
Andrew Yang:1.0
Tom Steyer:1.0
Amy Klobuchar:1.0
John Hickenlooper:1.0
Kirsten E. Gillibrand:1.0
John K. Delaney:1.0
Julián Castro:1.0
Cory A. Booker:1.0
Bill de Blasio:0.0
Marianne Williamson:0.0
Joe Sestak:0.0
Tim Ryan:0.0
Seth Moulton:0.0
Wayne Messam:0.0
Jay Robert Inslee:0.0
Mike Gravel:0.0
Steve Bullock:0.0
Michael F. Bennet:0.0
```

# APR 9 OUTLINE

- **Continue linear probing implementation of a hash table**
- **Hashing in Java: Strings, hashCode, HashSets, HashMaps**
- **Applications of hash tables**
  - Document classification
  - Spellcheck

# APR 9 OUTLINE

- Continue linear probing implementation of a hash table
- Hashing in Java: Strings, hashCode, HashSets, HashMaps
- Applications of hash tables
  - Document classification
  - Spellcheck

# MAP ADT (MANY WAYS OF DEFINING)

`lookup(k)`: if the map `M` has an entry with key `k`, return its associated value; else, return null

`insert(k, v)`: insert entry `(k, v)` into the map `M`; if key `k` is not already in M, then return null; else, replace old value with `v` and return old value associated with `k`

`remove(k)`: if the map `M` has an entry with key `k`, remove it from `M` and return its associated value; else, return null

`size()`, `isEmpty()`

`entrySet()`: return an iterable collection of the entries in `M`

`keySet()`: return an iterable collection of the keys in `M`

`values()`: return an iterable collection of the values in `M`

# HASH FUNCTIONS AND TABLES

**A hash function $h$ maps a key to integers in a fixed interval $[0, N-1]$**

**$h(x) = x \% N$ is such a function for integers**

**$h(x)$ is the hash value of key $x$**

**A hash table is an array of size $N$**

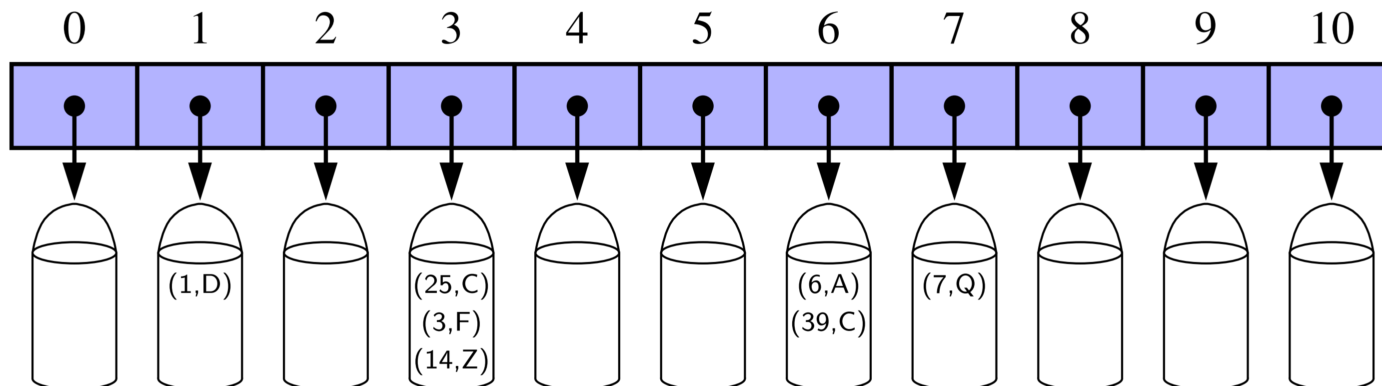
- associated hash function $h$
- item $(k, v)$ is stored at index $h(k)$

# COLLISION HANDLING 1: CHAINING

A hash function does not guarantee one-to-one mapping – no hash function does

One option: when more than one key hashes to the same index, we have a "bucket"

Each index holds a collection of entries



# COLLISION HANDLING 2: PROBING

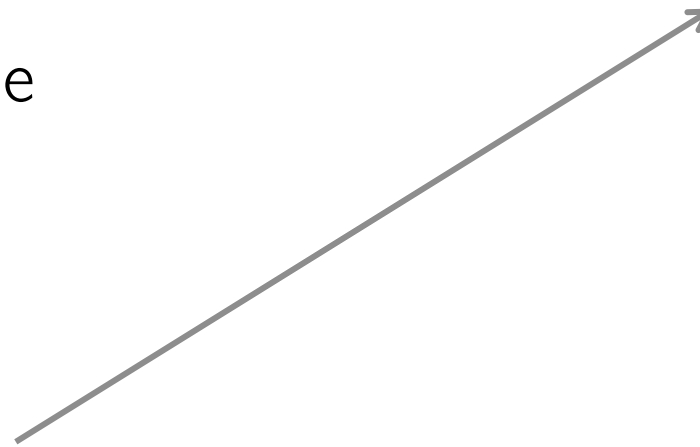
Colliding item is put in a different cell

**Example:** $h(x)=x\%13$

**insert 18, 41, 22, 44, 59, 32, 31, 73**

| | | 41 | | | 18 | 44 | 59 | 32 | 22 | 31 | 73 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

Linear probing: place the colliding item in the next (circularly) available table cell



Colliding items cluster together

# PROBING VS CHAINING

Probing is significantly faster in practice

Locality of references – much faster to access a series of elements in an array than to follow the same number of pointers in a linked list

# HASH TABLE IMPLEMENTATION

Say we have a very small hash table that contains the ID numbers of TAs, mapped to their name.

```
Table hashTable = new Table(5);
hashTable.insert(31,"Juvia");
hashTable.insert(13,"Steve");
hashTable.insert(100,"Will");
hashTable.insert(75,"Gareth");
hashTable.insert(28,"Lizzie");
```

Draw what the hash table will look like after the above code is executed, assuming we use a hash function that takes the key mod the length of the array.

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| null | null | null | null | null |

# HASH TABLE IMPLEMENTATION

Say we have a very small hash table that contains the ID numbers of TAs, mapped to their name.

```
Table hashTable = new Table(5);
hashTable.insert(31,"Juvia");
hashTable.insert(13,"Steve");
hashTable.insert(100,"Will");
hashTable.insert(75,"Gareth");
hashTable.insert(28,"Lizzie");
```

31 % 5 = 1

Draw what the hash table will look like after the above code is executed, assuming we use a hash function that takes the key mod the length of the array.

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| null | 31<br>Juvia | null | null | null |

# HASH TABLE IMPLEMENTATION

Say we have a very small hash table that contains the ID numbers of TAs, mapped to their name.

```
Table hashTable = new Table(5);
hashTable.insert(31,"Juvia");
hashTable.insert(13,"Steve");
hashTable.insert(100,"Will");
hashTable.insert(75,"Gareth");
hashTable.insert(28,"Lizzie");
```

31 % 5 = 1
13 % 5 = 3

Draw what the hash table will look like after the above code is executed, assuming we use a hash function that takes the key mod the length of the array.

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| null | 31<br>Juvia | null | 13<br>Steve | null |

# HASH TABLE IMPLEMENTATION

Say we have a very small hash table that contains the ID numbers of TAs, mapped to their name.

```
Table hashTable = new Table(5);
hashTable.insert(31,"Juvia");
hashTable.insert(13,"Steve");
hashTable.insert(100,"Will");
hashTable.insert(75,"Gareth");
hashTable.insert(28,"Lizzie");
```

31 % 5 = 1
13 % 5 = 3
100 % 5 = 0

Draw what the hash table will look like after the above code is executed, assuming we use a hash function that takes the key mod the length of the array.

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 100<br>Will | 31<br>Juvia | null | 13<br>Steve | null |

# HASH TABLE IMPLEMENTATION

Say we have a very small hash table that contains the ID numbers of TAs, mapped to their name.

```
Table hashTable = new Table(5);
hashTable.insert(31,"Juvia");
hashTable.insert(13,"Steve");
hashTable.insert(100,"Will");
hashTable.insert(75,"Gareth");
hashTable.insert(28,"Lizzie");
```

31 % 5 = 1
13 % 5 = 3
100 % 5 = 0
75 % 5 = 0

Draw what the hash table will look like after the above code is executed, assuming we use a hash function that takes the key mod the length of the array.

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 100<br>Will | 31<br>Juvia | 75<br>Gareth | 13<br>Steve | null |

# HASH TABLE IMPLEMENTATION

Say we have a very small hash table that contains the ID numbers of TAs, mapped to their name.

```
Table hashTable = new Table(5);
hashTable.insert(31,"Juvia");
hashTable.insert(13,"Steve");
hashTable.insert(100,"Will");
hashTable.insert(75,"Gareth");
hashTable.insert(28,"Lizzie");
```

31 % 5 = 1
13 % 5 = 3
100 % 5 = 0
75 % 5 = 0
28 % 5 = 3

Draw what the hash table will look like after the above code is executed, assuming we use a hash function that takes the key mod the length of the array.

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 100<br>Will | 31<br>Juvia | 75<br>Gareth | 13<br>Steve | 28<br>Lizzie |

# EXAMPLE IMPLEMENTATION WITHOUT GENERICS

```
private class TableRow {
    int key;
    String value;
    TableRow(int key, String value) {
        this.key = key;
        this.value = value;
    }
}
```

# EXAMPLE IMPLEMENTATION

```
public class Table {
    private TableRow[] rows;

    public Table(int tableSize) {
        rows = new TableRow[tableSize];
    }
```

# EXAMPLE IMPLEMENTATION

```
public class Table {
    private TableRow[] rows;

    public Table(int tableSize) {
        rows = new TableRow[tableSize];
    }

    // TODO 1: return the default position (index) where this key is stored
    private int hash(int key) {
        return key % rows.length;
    }

    // TODO 2: locates the position (index) where the specified key can be found,
    // or where it should be inserted if it is not already in the table
    private int locate(int key) {
```

# EXAMPLE IMPLEMENTATION

```
public class Table {
    private TableRow[] rows;

    public Table(int tableSize) {
        rows = new TableRow[tableSize];
    }

    // TODO 1: return the default position (index) where this key is stored
    private int hash(int key) {
        return key % rows.length;
    }

    // TODO 2: locates the position (index) where the specified key can be found,
    // or where it should be inserted if it is not already in the table
    private int locate(int key) {
        int pos = hash(key);
        while (row[pos] != null && row[pos].key != key) {
            pos = (pos + 1) % rows.length;
        }
        return pos;
```

linear probing (circular)

# EXAMPLE IMPLEMENTATION

*Hint: use locate!*

```
// TODO 3: put the specified value in the table under the specified key
public void insert(int key, String value) {







}

// TODO 4: retrieve the value associated with the given key
public String lookup(int key) {




}
```

*If finished: think about runtime (expected and worst case)*

# EXAMPLE IMPLEMENTATION

*Hint: use locate!*

```
// TODO 3: put the specified value in the table under the specified key
public void insert(int key, String value) {
        int pos = locate(key)
        TableRow newRow = new TableRow(key, value)
          if rows[pos] == null:
             rows[pos] = newRow
             return null
          else:
             rows[pos] = newRow
             value = rows[pos].value
             return value
}
// TODO 4: retrieve the value associated with the given key
public String lookup(int key) {
        int pos = locate(key)

        return rows[pos].value
}
```

string

hash → may not be free

case for null

*If finished: think about runtime (expected and worst case)*

# HASH TABLE RUNTIMES

**Let *n* be the number of elements in the hash table**

| | Hash Expected | Hash Worst |
|---|---|---|
| lookup | | |
| insert | | |
| remove | | |
| min/max | | |

Mute Stop Video Security Participants 27 Polling New Share Pause Share Annotate Remote Control More
ID: 483-963-860 Stop Share
Mouse Select Text Draw Stamp Spotlight Eraser Format Undo Redo Clear Save

# HASH TA

**Let *n* be the number of elements in the hash table**

| | Hash Expected | Hash Worst |
|---|---|---|
| lookup | O(1) | O(n) |
| insert | O(1) | O(n) |
| remove | O(1) | O(n) |
| min/max | O(n) | O(n) |

value

# HASH TABLE RUNTIMES

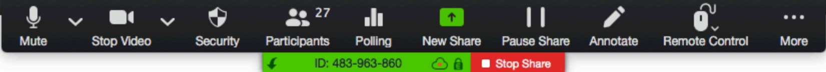
**Let *n* be the number of elements in the hash table**

| | Hash Expected | Hash Worst |
|---|---|---|
| lookup | $O(1)$ | $O(n)$ |
| insert | $O(1)$ | $O(n)$ |
| remove | $O(1)$ | $O(n)$ |
| min/max | $O(n)$ | $O(n)$ |

# APR 9 OUTLINE

- Continue linear probing implementation of a hash table
- **Hashing in Java: Strings, hashCode, HashSets, HashMaps**
- Applications of hash tables
  - Document classification
  - Spellcheck

# OTHER HASH FUNCTIONS FOR INTEGERS





# OTHER HASH FUNCTIONS FOR INTEGERS

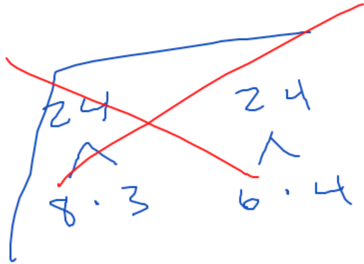

N = array length

key

$\star \quad h(x) = x \% N$

$h(x) = (ax + b) \% N$

prime { a → spread out range; b → shift }



# HASH CODE FOR STRINGS

We said before that the key could be flexible, but the hash function must return an **int** (so we can use it to index into the array)

Here is a partial table of ascii values for characters

| Dec | Char | Dec | Char | Dec | Char |
|---|---|---|---|---|---|
| 32 | SPACE | 64 | @ | 96 | ` |
| 33 | ! | 65 | A | 97 | a |
| 34 | " | 66 | B | 98 | b |
| 35 | # | 67 | C | 99 | c |
| 36 | $ | 68 | D | 100 | d |
| 37 | % | 69 | E | 101 | e |
| 38 | & | 70 | F | 102 | f |
| 39 | ' | 71 | G | 103 | g |
| 40 | ( | 72 | H | 104 | h |
| 41 | ) | 73 | I | 105 | i |
| 42 | * | 74 | J | 106 | j |
| 43 | + | 75 | K | 107 | k |
| 44 | , | 76 | L | 108 | l |
| 45 | - | 77 | M | 109 | m |
| 46 | . | 78 | N | 110 | n |
| 47 | / | 79 | O | 111 | o |
| 48 | 0 | 80 | P | 112 | p |
| 49 | 1 | 81 | Q | 113 | q |
| 50 | 2 | 82 | R | 114 | r |
| 51 | 3 | 83 | S | 115 | s |
| 52 | 4 | 84 | T | 116 | t |
| 53 | 5 | 85 | U | 117 | u |
| 54 | 6 | 86 | V | 118 | v |
| 55 | 7 | 87 | W | 119 | w |
| 56 | 8 | 88 | X | 120 | x |
| 57 | 9 | 89 | Y | 121 | y |
| 58 | : | 90 | Z | 122 | z |
| 59 | ; | 91 | [ | 123 | { |
| 60 | < | 92 | \ | 124 | \| |
| 61 | = | 93 | ] | 125 | } |
| 62 | > | 94 | ^ | 126 | ~ |
| 63 | ? | 95 | _ | 127 | DEL |

Credit: Richard E. Pattis (CMU)

# HASH CODE FOR STRINGS

Takeaway: hashCode is another way to write the key (in number form)

HASH CODE FOR STRINGS

Java hashcode

String s = "hello";

ascii

$$hashCode(s) = s[0] \cdot 31^{n-1} + s[1] \cdot 31^{n-2} + \cdots s[n-1]$$

$n = len(s)$

$$hashcode(\text{"hello"}) = 104 \cdot 31^4 + 101 \cdot 31^3 + 108 \cdot 31^2 + 108 \cdot 31^1 + 111$$

don't *mod* here because the hash table implementation will do that for us

# JAVA HASHSET AND HASHMAP

- Java has its own built-in hash tables and sets
- A hash set is kind of like a hash map, but just contains keys

```
HashSet<String> words = new HashSet<String>();
words.add("summer");
words.add("spring");
words.add("spring");

System.out.println(words.size());
System.out.println(words);
```

# JAVA HASHSET AND HASHMAP

- Java has its own built-in hash tables and sets
- A hash set is kind of like a hash map, but just contains keys

```
HashSet<String> words = new HashSet<String>();
words.add("summer");
words.add("spring");
words.add("spring");

System.out.println(words.size());
System.out.println(words);
```

```
2
[spring, summer]
```

# JAVA HASHSET AND HASHMAP

- Java has its own built-in hash tables and sets
- A hash set is kind of like a hash map, but just contains keys

```
HashMap<Integer, String> months = new HashMap<Integer, String>();
months.put(5, "april");
months.put(5, "may");
months.put(4, "april");

System.out.println(months.size());
System.out.println(months);
```

# JAVA HASHSET AND HASHMAP

- Java has its own built-in hash tables and sets
- A hash set is kind of like a hash map, but just contains keys

```
HashMap<Integer, String> months = new HashMap<Integer, String>();
months.put(5, "april");
months.put(5, "may");
months.put(4, "april");

System.out.println(months.size());
System.out.println(months);
```

```
2
{4=april, 5=may}
```

# OVERRIDING HASHCODE

Say we have a simple class with a few instance variables like the one below:

```
public class Student {

    private String name;
    private int id;

    public Student(String name, int id) {
        this.name = name;
        this.id = id;
    }
}
```

# OVERRIDING HASHCODE

Main.java

```
public static void main(String[] args) {

    Student will1 = new Student("Will", 99);
    Student will2 = new Student("Will", 99);

    HashSet<Student> tas = new HashSet<Student>();
    tas.add(will1);
    tas.add(will2);

    System.out.println(will1.equals(will2));
    System.out.println(tas.size());
}
```

Output

# OVERRIDING HASHCODE

Q: what does equals compare if we don't override it?
A: It compares the memory addresses (which in this case would be different)

Main.java

```
public static void main(String[] args) {

    Student will1 = new Student("Will", 99);
    Student will2 = new Student("Will", 99);

    HashSet<Student> tas = new HashSet<Student>();
    tas.add(will1);
    tas.add(will2);

    System.out.println(will1.equals(will2));
    System.out.println(tas.size());
}
```

<u>Output</u>

false
2

# OVERRIDING HASHCODE

Student.java

```
@Override
public boolean equals(Object o) {
    if (!(o instanceof Student)) {
        return false;
    }

    Student other = (Student) o;
    return name.equals(other.name) && id == other.id;
}
```

Main.java

```
public static void main(String[] args) {

    Student will1 = new Student("Will", 99);
    Student will2 = new Student("Will", 99);

    HashSet<Student> tas = new HashSet<Student>();
    tas.add(will1);
    tas.add(will2);

    System.out.println(will1.equals(will2));
    System.out.println(tas.size());
}
```

Output

# OVERRIDING HASHCODE

Student.java

```java
@Override
public boolean equals(Object o) {
    if (!(o instanceof Student)) {
        return false;
    }

    Student other = (Student) o;
    return name.equals(other.name) && id == other.id;
}
```

Main.java

```java
public static void main(String[] args) {

    Student will1 = new Student("Will", 99);
    Student will2 = new Student("Will", 99);

    HashSet<Student> tas = new HashSet<Student>();
    tas.add(will1);
    tas.add(will2);

    System.out.println(will1.equals(will2));
    System.out.println(tas.size());
}
```

Output

true
2

# OVERRIDING HASHCODE

Student.java

```
@Override
public int hashCode() {
    return name.hashCode() + id;
}
```

Main.java

```
public static void main(String[] args) {

    Student will1 = new Student("Will", 99);
    Student will2 = new Student("Will", 99);

    HashSet<Student> tas = new HashSet<Student>();
    tas.add(will1);
    tas.add(will2);

    System.out.println(will1.equals(will2));
    System.out.println(tas.size());
}
```

<u>Output</u>

# OVERRIDING HASHCODE

Student.java

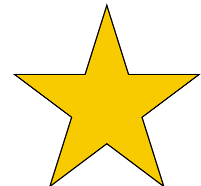
```
@Override
public int hashCode() {
    return name.hashCode() + id;
}
```

Main.java

```
public static void main(String[] args) {

    Student will1 = new Student("Will", 99);
    Student will2 = new Student("Will", 99);

    HashSet<Student> tas = new HashSet<Student>();
    tas.add(will1);
    tas.add(will2);

    System.out.println(will1.equals(will2));
    System.out.println(tas.size());
}
```

<u>Output</u>

true
1



# APR 9 OUTLINE

- Continue linear probing implementation of a hash table
- Hashing in Java: Strings, hashCode, HashSets, HashMaps
- **Applications of hash tables**
  - Document classification
  - Spellcheck

# GROUP EXERCISE

Think of the high-level algorithm first, then pseudocode. Assume you can easily read in words from the user or from a document.

1) **Word frequencies**: use a hash table to count the frequencies of each word in a document. This is very useful for document classification, which is (in part) how Google search works!

2) **Spellchecker**: assume you have a HashSet of “correctly” spelled words. Given a word from the user, how could you tell if it is spelled correctly? How could you go about suggesting close alternatives if it’s misspelled?

We will discuss these more next time!