# CS Responsibility

## Are Political Ads Free Speech?

- Facebook allows advertisers to run ads about politics, elections, and other social issues
  - “Free expression”(Zuckerberg, from Facebook post)
  - “I don’t think it’s right for private companies to censor politicians and the news”(Zuckerberg, Washington Post)
  - “Ads can be an important part of voice -- especially for candidates and advocacy groups the media might not otherwise cover so they can get their message into debates. And it's hard to define where to draw the line”(Zuckerberg, from Facebook post)
- Twitter recently announced that it will ban all political ads starting November
  - “Reach should be earned, not bought”(Dorsey, Vox)
  - “This isn’t about free expression. This is about paying for reach. And paying to increase the reach of political speech has significant ramifications that today’s democratic infrastructure may not be prepared to handle. It’s worth stepping back in order to address” (Dorsey, Vox)


Sources:
https://www.facebook.com/zuck/posts/10110264733792991
https://www.washingtonpost.com/technology/2019/10/17/zuckerberg-standing-voice-free-expression/
https://www.vox.com/recode/2019/10/30/20940612/twitter-political-ads-announcement-jack-dorsey-facebook

# Free Speech: Some Context

- First Amendment in the Bill of Rights:
  - **Congress** shall make **no law** respecting an establishment of religion or prohibiting the free exercise thereof; or **abridging the freedom of speech**, or of **the press**, or the right of the people peaceably to assemble, and to petition the Government for a redress of grievances.
- Rules on social media platforms do not have to comply with the first amendment
  - Legal responsibilities vs. social values
  - Self-regulation already exists!
    - Child pornography
    - Two-pass system: algorithm -> human
- Zuckerberg's power to curate your news!
  - 2 Billion+ users
- What level of harm calls for a limits to "free expression"? These answers are relevant to the algorithms we use to filter out inappropriate content

# Consequences

**Facebook**:

- Zuckerberg explicitly permits political ads with known lies
- Outrage spreads faster than facts
- Echo chambers and confirmation bias
- Use of third-party fact checking
  - disinformation
- Past ban in WA state backfired

**Twitter**:

- What should and shouldn't be considered political?
  - "issue ads"
- Very hard to regulate

**Which opinion appeals to you more and why?**

Sources: https://www.theverge.com/2019/10/31/20941917/twitter-political-ads-ban-facebook-washington-state-bob-ferguson
https://www.cnbc.com/2019/11/03/facebook-and-twitter-get-it-wrong-when-it-comes-to-political-ads.html

# Hashing

## Sets and Maps

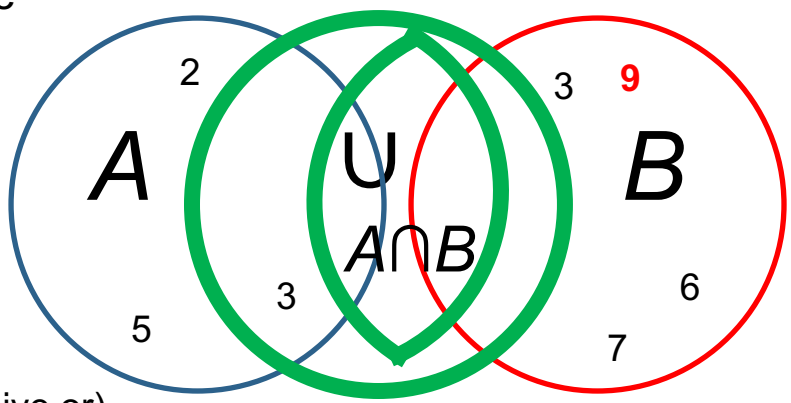
# So Far …

- Covered some Abstract Data Types (ADTs) which store a collection of objects (Stacks, Queues, Trees), and a variety of ways to implement them (arrays, linked lists)

- Now cover another ADT which stores a collection, called a Set

# Introducing… Sets

- A set is a collection of unique, unordered elements
  - no duplicates
  - A == {2,3,5} == {5,3,2}
  - A, B can be single elements or sets of multiple elements
- Basic operations of the Set ADT:
  - add element to set
  - remove element from set
  - check if element is in set
  - Union: merge two sets together
    - ex: Union set contains students who are CS15 students or graduate students (or both – inclusive or)
  - Intersection: Intersection set contains only elements in two sets that are in both
    - ex: students who are both CS15 students and graduate students



# Set Abstract Data Type (1/2)

- Sets can be implemented using arrays, lists, hashing (slide 29), etc.
- No indices, no random access
- Useful for:
  - checking if elements of one collection are also a part of another collection (e.g., finding all students in CS15 who are also taking ECON0100). Since there is no explicit intersection operator in Java, we must loop through the elements of the smaller set, and check membership in the larger set
  - prevent an array from storing duplicates by checking an element to be inserted against a set of previously encountered names: if it is already in the set, it is a duplicate, if not, enter it into array and set. The win is in the efficiency of checking if an element is in a set (O(1)) versus having to search for it in the array (O(N))

# Set Abstract Data Type (2/2)

- Because there is no order/index, `Sets` can be implemented differently than `Lists` and other ADTs we have shown so far
- Java has a class `java.util.HashSet<Type>` specialized for set operations. This class implements the `Set` interface and is backed by a `Hash Table`.

# HashSet Methods (1/2)

```
/*Constructor returns new HashSet capable of holding elements of type Type.
 *Java will let us create non-homogeneous sets, but we rarely want this, so
  specify use the generic Type to enforce homogeneity */
public HashSet<Type> HashSet<Type>()

/*adds element e to HashSet, if not already present (returns false if
element is already present)*/
public boolean add(Type e)

/*returns true if this set contains the specified element.
 *note on parameter type: Java accepts any Object since the elements of
 *your set could be any object, but you should supply one of type Type
  for good programming practices */
public boolean contains(Object o)
```

# HashSet Methods (2/2)

```
//removes all elements from this set
public void clear()

//returns true if this set contains no elements
public boolean isEmpty()

/*removes specified element from this set if present
 *note on parameter type: Java accepts any Object since the elements of
  your set could be any object, but you should supply one of type Type*/
public boolean remove(Object o)

//returns the number of elements in this set
public int size()

//see JavaDocs for more methods
```

# Iteration over a HashSet

- You can also iterate over elements stored in a `HashSet` by using a `for-each` loop.
  - as it is a set, there is no guaranteed order of processing elements

```
HashSet<String> strings = new HashSet<String>();

//elided adding elements to the set

for (String s:strings) {//in HashSet strings, of type String, for each element s
   System.out.println(s); //prints all Strings in HashSet
}
```

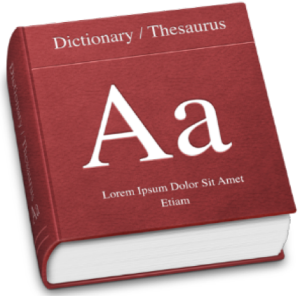
# HashSet Example

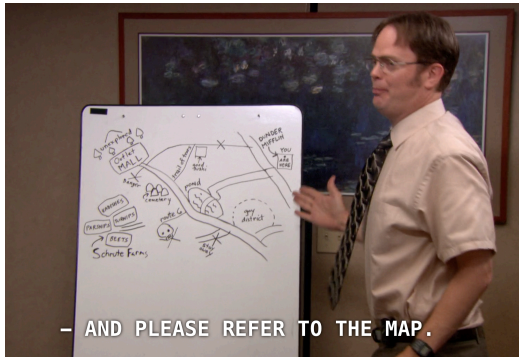
```
//somewhere in your app
HashSet<String> springCourses = new HashSet<String>();
springCourses.add(“BIOL0200”);
springCourses.add(“ECON0110”);
//elided adding rest of Banner

//in another part of your program
if (springCourses.contains(“CS0160”){
    System.out.println(“I can take cs16 next semester!”);
}
//elided checking for other classes
```

As we will see, each check for set membership takes just O(1)! i.e., no actual searching!

# Introducing… Maps (1/3)

- Maps are used to store (key, value) pairs.
  - a key is used to lookup its corresponding value
- (Word, Definition) in a dictionary
- (Brown ID, Person) in Banner
- (Name, Phone #) in a contacts list
- (Identifier, Memory address) in compiler–called symbol table
- Think of a map as discrete function that maps from domain to co-domain

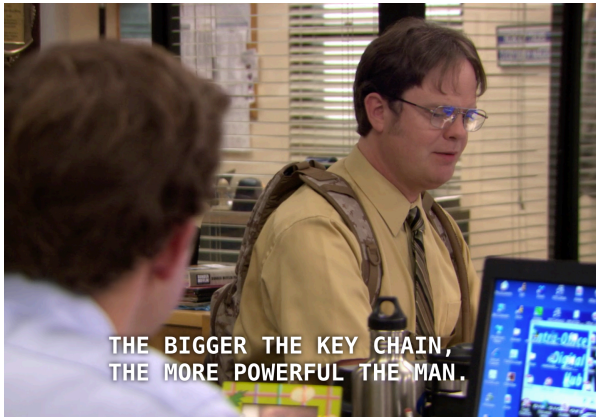


# Introducing… Maps (2/3)

- Java provides `java.util.HashMap<K,V>` class
- Often called a “hash table”
- Other structures that provide maps include `TreeMap` , `Hashtable`, `LinkedHashMap`, and more
  - each has its own advantages and drawbacks
  - we will focus on `HashMap`
- `HashMap`s have constant time insert, removal, and search!–explained shortly

# HashMap Syntax




- Like other ADTs, need to specify type of elements it holds
- This time need to specify type of both key AND value
- Key and value can be instances of any class

```
new HashMap<KeyClass, ValueClass>();
```

Note: In this case the "<>" are literal to indicate generics; you must type them when writing your own code

- Only one entry for a given key - no duplicates

# HashMap Syntax

- If we wanted to map an Integer to its String representation

```
HashMap<Integer, String> intTable = new HashMap<Integer, String>();
```

- If we wanted to map a TA to their Birthday

```
HashMap<CS15TA, Date> birthdayTable = new HashMap<CS15TA, Date>();
```

- In all cases, both key and value types must resolve to a type (e.g., class, interface)

- Note: Can't use `int` or `boolean` as a type because they are *primitives,* not classes
  - so use a built-in class that is equivalent to that primitive (wrapper), `Integer` or `Boolean` respectively

# java.util.HashMap Methods (1/2)

```
/*K refers to type of Key, V to type of value.
 Adds specified key, value pair to the table, returns value. If
    there already was an entry for this key, it is replaced*/
public V put(K key, V value)
/*returns value to which the specified key is mapped, or null
 *if map contains no mapping for the key.
 *note on parameter type: Java accepts any Object, but you should
  supply the same type as the key*/
public V get(Object key)

//returns the number of keys in this hashtable
public int size()
```

# java.util.HashMap Methods (2/2)

```
/*note on parameter type: Java accepts any Object, but you
 *should supply the same type as either the key or the value.
  Predicate tests if specified object is a key in this hash table*/
public boolean containsKey(Object key)

//returns true if hash table maps at least one key to this value
public boolean containsValue(Object Value)

/*removes key and its corresponding value from hash table,
  returns value which the key mapped to or null if key had no mapping */
public V remove(Object key)

//more methods in JavaDocs
```

# Finding out your friends' logins (1/4)

- Given an array of CS students who have the properties "csLogin" and "real name", how might you efficiently find out your friends' logins?

- Givens
  - `String[] _friends`, an array of your friends' names
  - `CSStudent[] _students`, an array of students with a "csLogin" and a "real name"

# Finding out your friends' logins (2/4)

- Old Approach:

```
for (int i=0; i < _friends.length; i++){ //for all friends
    for (int j=0; j < _students.length; j++){ //for all students
        if (_friends[i].equals(_students[j].getName())){ //getName() code elided
            String login = _students[j].getLogin(); //getLogin() code elided
            System.out.println(_friends[i] + "'s login is " + login + "!");
        }
    }
}
```

- Note: Use `String` class' `equals()` method because "`==`" checks for equality of reference, not of content
- This is $O(n^2)$–far from optimal

# Finding out your friends' logins (3/4)

- Better solution: use a `HashMap` to store students instead of an array:
  - key is name
  - value is login
  - use name to look up login!

# Finding out your friends' logins (4/4)

- Using a HashMap

```
HashMap<String, String> myTable = new HashMap<String, String>();
for (CSStudent student : _students){ //same array of students
    //getName() and getLogin() code elided
    myTable.put(student.getName(), student.getLogin()); //build HashMap
}
for (String friendName : _friends){ //same array of friends
    String login = myTable.get(friendName); //look up friend's login

    if (login == null){
        System.out.println("No login found for " + friendName);
        continue;
    }
    System.out.println(friendName + "'s login is " + login + "!");
}
```

- Each insert and search in HashMap is O(1)

# TopHat Question

What is the runtime of the function to find all of your friends' logins now that we're using a `HashMap`? (assume a large number of friends…)

```
HashMap<String, String> myTable = new HashMap<String, String>();
for (CSStudent student : _students){
    myTable.put(student.getName(), student.getLogin());
}
for (String friendName : _friends){
    String login = myTable.get(friendName);

    if (login == null){
        System.out.println("No login found for " + friendName);
        continue;
    }
    System.out.println(friendName + "'s login is " + login + "!");
}
```

A. O(1)
B. O(n)
C. O(log(n))
D. $O(n^2)$

# Counting frequency in an Array (1/4)

- How many times does a given word show up in a given string?
- Consider a book as one long `String`. That's too hard to search, so let's chop the string into individual words using punctuation as a separator and put each word in an array

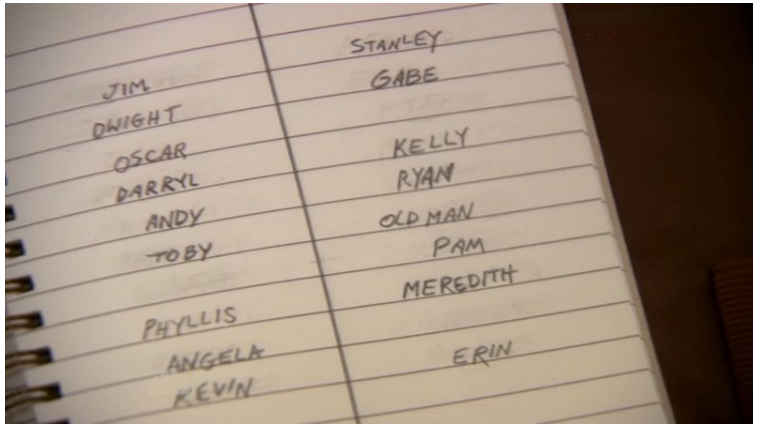
- Givens
  - `String[] _book`, an array of `String`s, each an individual word
  - `String searchTerm`, the word you're looking for

# Counting frequency in an Array (2/4)

```
public void frequency(String searchTerm) {
    int wordCounter = 0; //frequency of single term
    for (String word : _book){
        if (word.equals(searchTerm)){
            wordCounter++;
        }
    }
    System.out.println(searchTerm + " appears " +
     wordCounter + " times");
}
```

# Counting frequency in an Array (3/4)

- When tracking one word, code is simple
- But what if we wanted to keep track of 5 words? 100?
- Should we make instance variables to count the frequency of each word? For each term in the book?
  - should we iterate through _book for each of the search terms? Sounds like $O(n^2)$...



# Counting frequency in an Array (4/4)

```
HashMap<String, Integer> countMap = new HashMap<String, Integer>();
/*_book is an array of words.
 If currWord in _book matches a search term,
 put currWord back with updated count. By using
 put(), we replace current entry in hashMap.
 Note use of Integer rather than int because you
 can't use base types as generics */
for (String currWord : _book){
    if (countMap.containsKey(currWord){
        Integer count = countMap.get(currWord);
        count++;
        countMap.put(currWord, count);
    }
    else{
        //First time seeing word
        countMap.put(currWord, 1);
    }
}
```

```
/*separate method: searchTerms is now an array of
Strings we're counting */
public void frequencies(String[] searchTerms) {
    for (String word : searchTerms){
       Integer freq = countMap.get(word);
       if (freq == null){
          freq = 0;
        }
        System.out.println(word + " shows up " +
           freq + " times!");
     }
}
```

Despite increase in search terms, still O(n)

# Map Implementation (1/4)

- How do we implement a Map with constant-time insertion, removal, and search?
- In essence, we are searching through a data structure for value associated with key
  - similar to searching problem we have been trying to optimize
- Searching in an array:
  - unsorted array is O(n)
  - sorted array is O(log n), as is tree
    - remember binary partitioning of array in merge sort where tree depicting passes, and binary search tree both had depth of $\log_2 n$?
  - can we do even better than $\log_2 n$?!? That would be O(1)!!!
    - yes: with hashing, but has limitations–look back at Trees lecture

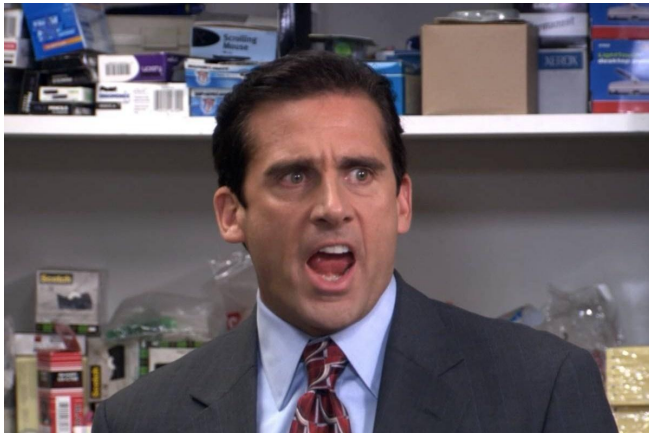
# Map Implementation (2/4)

- Try a radically different approach, using an array
- What if we could directly use the key as an index to access appropriate spot in the array?
- Remember: digits, alphanumerics, symbols, even control characters are all stored as bit strings–"it's bits all the way down…"
  - see ASCII table
  - bit strings can be interpreted as numbers in binary that can be used to index into an array to get oct or hex equivalent
  - O(1) to find the key in array at given index!!!

| Char | Dec | Oct | Hex | Char | Dec | Oct | Hex | Char | Dec | Oct | Hex |
|---|---|---|---|---|---|---|---|---|---|---|---|
| (sp) | 32 | 0040 | 0x20 | @ | 64 | 0100 | 0x40 | ` | 96 | 0140 | 0x60 |
| ! | 33 | 0041 | 0x21 | A | 65 | 0101 | 0x41 | a | 97 | 0141 | 0x61 |
| " | 34 | 0042 | 0x22 | B | 66 | 0102 | 0x42 | b | 98 | 0142 | 0x62 |
| # | 35 | 0043 | 0x23 | C | 67 | 0103 | 0x43 | c | 99 | 0143 | 0x63 |
| $ | 36 | 0044 | 0x24 | D | 68 | 0104 | 0x44 | d | 100 | 0144 | 0x64 |
| % | 37 | 0045 | 0x25 | E | 69 | 0105 | 0x45 | e | 101 | 0145 | 0x65 |
| & | 38 | 0046 | 0x26 | F | 70 | 0106 | 0x46 | f | 102 | 0146 | 0x66 |
| ' | 39 | 0047 | 0x27 | G | 71 | 0107 | 0x47 | g | 103 | 0147 | 0x67 |
| ( | 40 | 0050 | 0x28 | H | 72 | 0110 | 0x48 | h | 104 | 0150 | 0x68 |
| ) | 41 | 0051 | 0x29 | I | 73 | 0111 | 0x49 | i | 105 | 0151 | 0x69 |
| * | 42 | 0052 | 0x2a | J | 74 | 0112 | 0x4a | j | 106 | 0152 | 0x6a |
| + | 43 | 0053 | 0x2b | K | 75 | 0113 | 0x4b | k | 107 | 0153 | 0x6b |
| , | 44 | 0054 | 0x2c | L | 76 | 0114 | 0x4c | l | 108 | 0154 | 0x6c |
| - | 45 | 0055 | 0x2d | M | 77 | 0115 | 0x4d | m | 109 | 0155 | 0x6d |
| . | 46 | 0056 | 0x2e | N | 78 | 0116 | 0x4e | n | 110 | 0156 | 0x6e |
| / | 47 | 0057 | 0x2f | O | 79 | 0117 | 0x4f | o | 111 | 0157 | 0x6f |
| 0 | 48 | 0060 | 0x30 | P | 80 | 0120 | 0x50 | p | 112 | 0160 | 0x70 |
| 1 | 49 | 0061 | 0x31 | Q | 81 | 0121 | 0x51 | q | 113 | 0161 | 0x71 |
| 2 | 50 | 0062 | 0x32 | R | 82 | 0122 | 0x52 | r | 114 | 0162 | 0x72 |
| 3 | 51 | 0063 | 0x33 | S | 83 | 0123 | 0x53 | s | 115 | 0163 | 0x73 |
| 4 | 52 | 0064 | 0x34 | T | 84 | 0124 | 0x54 | t | 116 | 0164 | 0x74 |
| 5 | 53 | 0065 | 0x35 | U | 85 | 0125 | 0x55 | u | 117 | 0165 | 0x75 |
| 6 | 54 | 0066 | 0x36 | V | 86 | 0126 | 0x56 | v | 118 | 0166 | 0x76 |
| 7 | 55 | 0067 | 0x37 | W | 87 | 0127 | 0x57 | w | 119 | 0167 | 0x77 |
| 8 | 56 | 0070 | 0x38 | X | 88 | 0130 | 0x58 | x | 120 | 0170 | 0x78 |
| 9 | 57 | 0071 | 0x39 | Y | 89 | 0131 | 0x59 | y | 121 | 0171 | 0x79 |
| : | 58 | 0072 | 0x3a | Z | 90 | 0132 | 0x5a | z | 122 | 0172 | 0x7a |
| ; | 59 | 0073 | 0x3b | [ | 91 | 0133 | 0x5b | { | 123 | 0173 | 0x7b |
| < | 60 | 0074 | 0x3c | \ | 92 | 0134 | 0x5c | \| | 124 | 0174 | 0x7c |
| = | 61 | 0075 | 0x3d | ] | 93 | 0135 | 0x5d | } | 125 | 0175 | 0x7d |
| > | 62 | 0076 | 0x3e | ^ | 94 | 0136 | 0x5e | ~ | 126 | 0176 | 0x7e |
| ? | 63 | 0077 | 0x3f | _ | 95 | 0137 | 0x5f | | | | |

# Map Implementation (3/4)

- But creating an array to look up CS15 students (value) based on Banner ID # (key) would be a *tremendous* waste of space
  - if ID number is one letter followed by eight digits (e.g., B00011111), there are $10^8$ combinations!
  - do not want to allocate 100,000,000 words for no more than 400 students
  - (1 word = 4 bytes)
  - array would be terribly sparse…
- What about using social security number?
  - would need to allocate $10^9$ words, about 4 gigabytes, for no more than 400 students! And think about arbitrary names <30 chars: need $26^{30}$ !!

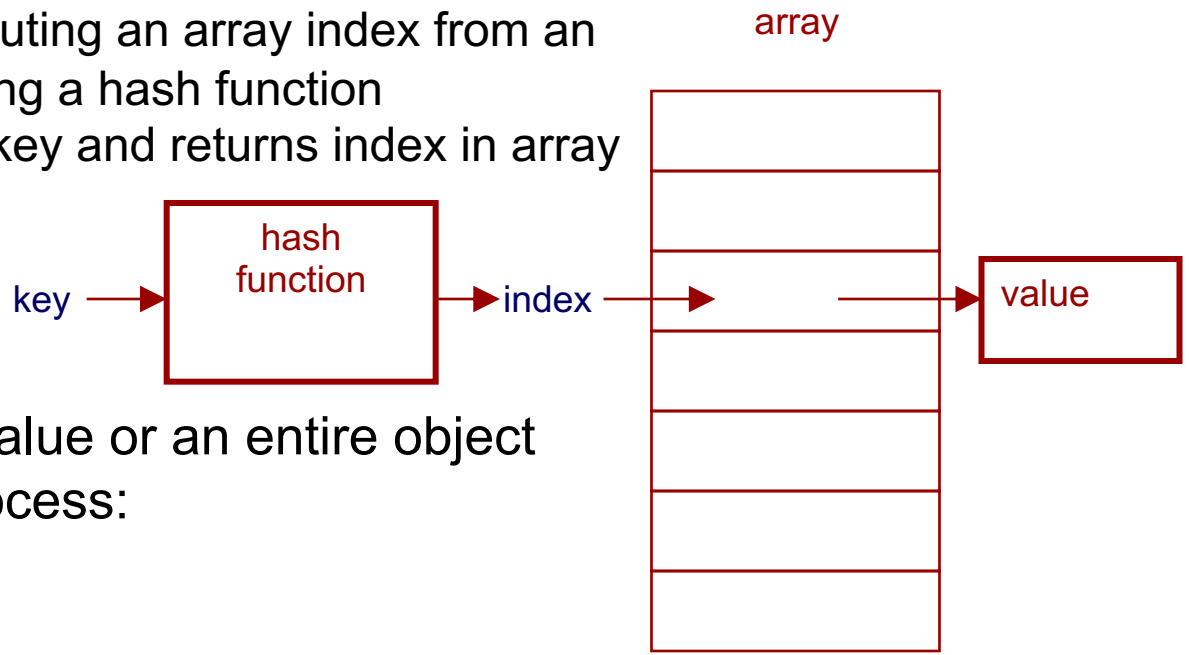


# Map Implementation (4/4)

- Thus, two major problems:
  - how can we deal with arbitrarily long keys, both numeric *and* alphanumeric?
  - how can we build a small, dense (i.e., space-efficient) array that we can index into to find keys and values?
- Impossible?
  - No, we approximate

# Hashing

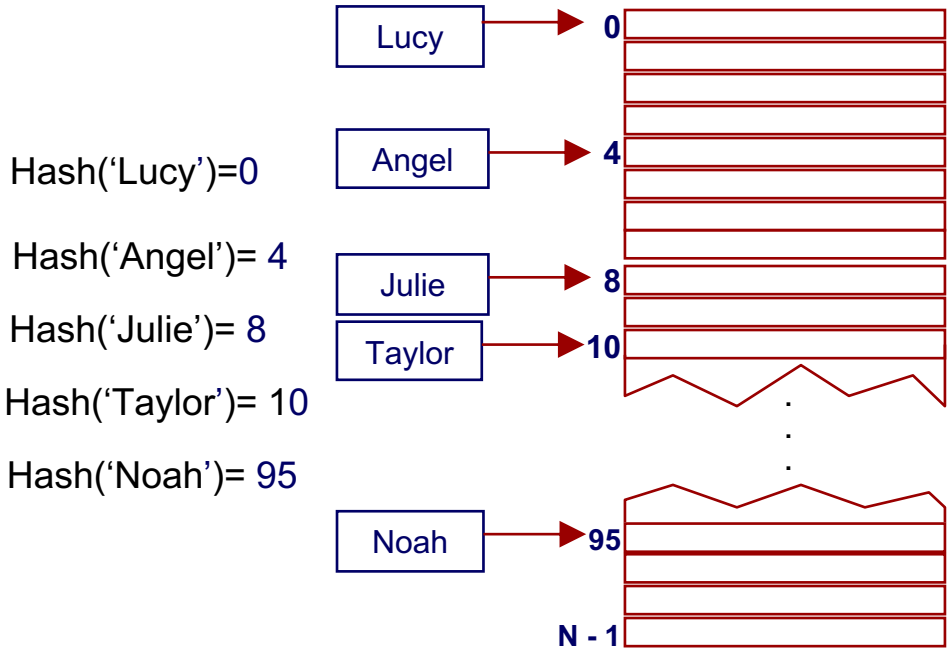
- How do we approximate?
  - we use hashing
  - hashing refers to computing an array index from an arbitrarily large key using a hash function
  - hash function takes in key and returns index in array




- Index leads to a simple value or an entire object
- Therefore, a two-step process:
  - hash to create index
  - use index to get value

# Hashing

- Array used in hashing typically holds several hundred to several thousand entries; size typically a prime (e.g., 1051)
  - array of links to instances of the class HTA



# Hash Functions (1/4)

- An example of a hash function for alphanumeric keys
  - ASCII is a bit representation that lets us represent all alphanumeric symbols as integers
  - take each character in key, convert to integer, sum integers—sum is index
  - but what if index is greater than array size?
  - use mod, i.e. (index % arrayLength) to ensure final index is in bounds
    - think as if index is being “wrapped around”
  - note: hash functions are non-reversible, meaning can’t get original data from output of hash function

# Hash Functions (2/4)

- Almost any reasonable function that uses all bits will do, so choose a fast one, and one that distributes more or less uniformly (randomly) in the array to minimize holes!
- A better hash function
  - take a string, chop it into sections of 4 letters each, then take value of 32 bits that make up each 4-letter section and XOR (exclusive OR) them together, then % (mod) that result by table size
- Will cover this more in CS16!

# Hash Functions (3/4)

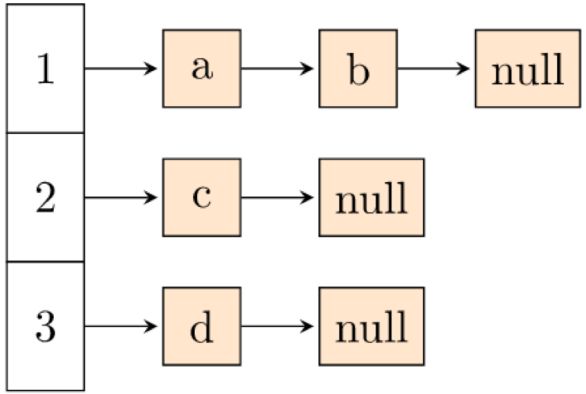
- We want to turn “noah korotzer” into an integer index for an array of size 101
  - Group into 4 character substrings
    - “noah” “koro” “tzer”
  - Turn each character into ASCII
    - 110 111 97 104 | 107 111 114 111 | 116 122 101 114
  - Turn each ASCII character into binary
  - 01101110 01101111 01100001 01101000 | 01101011 01101111 01110010 01101111 | 01110100 01111010 01100101 01110010

# Hash Functions (4/4)

- We want to turn “noah korotzer” into an integer index for an array of size 101
  - Turn each group into one value by mashing bits together
    - “noah” → 01101110011011110110000101101000
    - “koro” → 01101011011011110111001001101111
    - “tzer” → 01110100011110100110010101110010
  - XOR the 3 groups together
    - 01101110011011110110000101101000 ^ 01101011011011110111001001101111 ^ 01110100011110100110010101110010 = 11100010111101001110110011101 01
    - 11100010111101001110110011101 01 (binary) → 1903851125
  - Mod by size of list to ensure it’s within the array
    - Index = 1903851125 % 101 = 14

# Collisions (1/2)

- If we have 6,000 Brown student names that we are mapping to Banner IDs using an array of size 1051, clearly, we are going to get "collisions" where different keys will hash to the same index
- Does that kill the idea? No!
- Instead of having an array of type Value, we instead have each entry in the array be a `_head` pointer to an overflow "bucket" for all keys that hash to that index. The bucket can be, e.g., our perennial favorite, the unsorted singly linked list, or an array, whatever…
- So, if we get a collision, the linked list will hold all values with keys associated to that bucket



# Collisions (2/2)

- Since collisions are frequent, for methods like `get(key)` and `remove(key)`, `HashMap` will have to iterate through all items in the hashed bucket to `get` or `remove` the right object
- This is O(k), where k is the length of a bucket – it will be small, so brute force search is fine
- The best hash functions minimize collisions
- Java has its own efficient hash function, covered in CS16
- A way to think about hashing: a fast, large initial division (e.g., 1051-way), followed by a brute force search over a small bucket–even bucket size 100 is fast!

# TopHat Question

Given the following keys and hash functions:

| Key |
| --- |
| 0010 |
| 2040 |
| 3956 |
| 9754 |
| 2994 |

| Functions |
| --- |
| **f**(key) = key%2 |
| **g**(key) = key |
| **h**(key) = (sum of digits)%7 |

Which is the best hash function for the given data and why?

A. **f** because there are no unused buckets
B. **g** because there are no collisions
C. **h** because it minimizes collisions and space.
D. All 3 are good.

# HashMap Pseudocode

```
table = array of lists of some size
h = some hash function

public put(K key, V val):
    int index = h(key)
    table[index].addFirst(key, val)
```

O(1), if h( ) runs in O(1) time

```
public V get(K key):
    int index = h(key)
    /*search through (key, val) pairs
      in bucket at table[index] */
    for each (k, v) in table[index]:
        if k == key:
            return v
    return null //if not found, return null
```

Indexing with hash is O(1), and buckets are usually well under 100, so linear search time is trivial, O(1)

Note: LinkedLists only hold one element per node, so in actual code, use instance of a class that holds key and value

# HashMaps… efficiency for free?

- Not quite
- While `put()` and `get()` methods usually run in O(1) time, each takes more time than inserting at the end of a queue, for example
- A bit more memory expensive (array + buckets)
- Inefficient when many collisions occur (array too small)
- But it is likely the best solution overall, if you don't need order
  - (key, value) pairs are stored in random order based on hash. The best hash is random to minimize collisions.
  - in the Trees lecture, we showed that there are more complex queries for which Hash Table is very inefficient

# Hash Tables vs. Trees

- Hash Tables and Trees are different data structures used for different kinds of problems
- For just searching, insert/remove, a Hash Table will be faster
  - you know the exact key to search for
  - find a student's Banner ID given their name, key is name and value is Banner ID
- If you're trying to solve a nearest neighbors problem, a BST will be faster
  - you do not know the exact key to search for
  - find 4 people closest to a 95 in the class, key is grade and value is student name
- If you're trying to find the min and max in an array of numbers, a BST will be faster
- Can produce an already sorted list of data items by traversing the tree

# Announcements

- Tetris Early Deadline is **this Thursday**!
  - On-time deadline: Saturday, November 16
  - Late deadline: Monday, November 18

- Next week's lectures are important! Learn about the final projects and see demos.