So Far…
- Covered a variety of Abstract Data Types (ADTs) which store a collection of objects (Stacks, Queues, Lists), 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 elements or sets
- Basic Set Operations:
  - add element to set
  - remove element from set
  - merge two sets together (Union)
    - ex: all CS15 students and CS17 students
  - get elements in two sets which overlap (intersection)
    - ex: all CS15 students who are freshmen

Set Abstract Data Type (1/2)
- Sets can be implemented using arrays, lists, hashing (slide 28), 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 ECON 100)
  - keeping track of objects which meet some criteria (e.g., use sets: check an unordered array for duplicates by iterating over array, first checking if current element exists in set)
    - if YES, add to another array which holds duplicates for further processing (such as checking whether students registered for CS16 also took CS15)
    - if NO: add to the set
- Because there is no order/index, Sets can be implemented more efficiently than Lists and other ADTs we have shown so far
- Java has an implementation specialized for set operations, java.util.HashSet<Type>

HashSet Methods (1/2)
- Constructor returns new HashSet capable of holding elements of type Type
  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, but you should supply the same type as you are adding, using the <type>
  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, but you
//should supply the same type as you are adding, using the <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 an enhanced for loop.
  o as it is a set, there is no guaranteed order of elements over iteration

HashSet<String> strings = new HashSet<String>();
  //elided adding elements to the set
for (String s:strings) {
  //as it is a set, there is no guaranteed order of elements over iteration
  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 such 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, so 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

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
  o each has its own advantages and drawbacks
  o we will focus on HashMap
• HashMaps 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>();
**HashMap Syntax**
- If we wanted to map an `Integer` to its String representation:
  ```java
  HashMap<Integer, String> intTable = new HashMap<Integer, String>();
  ```
- If we wanted to map a `TA` to his/her `Birthday`:
  ```java
  HashMap<TA, Date> birthdayTable = new HashMap<TA, Date>();
  ```
- In all cases, both key and value types must resolve to a type (e.g., class, interface).
  - Note: Can't use `<int, boolean>` because both `int` and `boolean` are primitives, not classes.
    - cannot use a primitive type as a generic, so use a built-in class that is equivalent to that primitive (wrapper).
  - Instead use `<Integer, Boolean>`.

**java.util.HashMap Methods (1/2)**
- `public V put(K key, V value)`
  - adds specified key, value pair to the table.
  - 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.

**java.util.HashMap Methods (2/2)**
- `public boolean containsKey(Object key)`
  - tests if the specified object is a key in this hashtable.
  - public `boolean containsValue(Object value)`
  - returns true if hashtable maps at least one key to this value.
  - public `V remove(Object key)`
  - tests if the specified object is a key in this hashtable.
  - returns value which the key mapped to or `null` if key had no mapping.
  - 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?

  • Old Approach:
    ```java
    for (int i=0; i < _friends.length; i++) {
      for (int j=0; j < _students.length; j++) {
        if (_friends[i].equals(_students[j].getName())){
          String login = _students[j].getLogin();
          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 (2/4)**
- An approach using a `HashMap`:
  - `public V put(K key, V value)`
  - `public V get(Object key)`
  - `public boolean containsKey(Object key)`
  - `public boolean containsValue(Object Value)`
  - `public V remove(Object key)`

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

  • New Approach:
    ```java
    HashMap<String, String> loginTable = new HashMap<String, String>();
    for (int i=0; i < _students.length; i++) {
      loginTable.put(_students[i].getName(), _students[i].getLogin());
    }
    for (int i=0; i < _friends.length; i++) {
      if (loginTable.containsKey(_friends[i].getName())){
        String login = loginTable.get(_friends[i].getName());
        System.out.println(_friends[i] + "'s login is " + login + "!");
      }
    }
    ```
  - More efficient, $O(n)$ time complexity.
  - Note: If a key is not present, `get()` returns `null`.

**Finding out your friends’ logins (3/4)**
- An approach using a `HashMap`:
  - `public V put(K key, V value)`
  - `public V get(Object key)`
  - `public boolean containsKey(Object key)`
  - `public boolean containsValue(Object Value)`
  - `public V remove(Object key)`

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

  • New Approach:
    ```java
    HashMap<String, String> loginTable = new HashMap<String, String>();
    for (int i=0; i < _students.length; i++) {
      loginTable.put(_students[i].getName(), _students[i].getLogin());
    }
    for (int i=0; i < _friends.length; i++) {
      if (loginTable.containsKey(_friends[i].getName())){
        String login = loginTable.get(_friends[i].getName());
        System.out.println(_friends[i] + "'s login is " + login + "!");
      }
    }
    ```
  - More efficient, $O(n)$ time complexity.
  - Note: If a key is not present, `get()` returns `null`.

**Finding out your friends’ logins (4/4)**
- An approach using a `HashMap`:
  - `public V put(K key, V value)`
  - `public V get(Object key)`
  - `public boolean containsKey(Object key)`
  - `public boolean containsValue(Object Value)`
  - `public V remove(Object key)`

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

  • New Approach:
    ```java
    HashMap<String, String> loginTable = new HashMap<String, String>();
    for (int i=0; i < _students.length; i++) {
      loginTable.put(_students[i].getName(), _students[i].getLogin());
    }
    for (int i=0; i < _friends.length; i++) {
      if (loginTable.containsKey(_friends[i].getName())){
        String login = loginTable.get(_friends[i].getName());
        System.out.println(_friends[i] + "'s login is " + login + "!");
      }
    }
    ```
  - More efficient, $O(n)$ time complexity.
  - Note: If a key is not present, `get()` returns `null`.

**Finding out your friends’ logins (4/4)**
- An approach using a `HashMap`:
  - `public V put(K key, V value)`
  - `public V get(Object key)`
  - `public boolean containsKey(Object key)`
  - `public boolean containsValue(Object Value)`
  - `public V remove(Object key)`

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

  • New Approach:
    ```java
    HashMap<String, String> loginTable = new HashMap<String, String>();
    for (int i=0; i < _students.length; i++) {
      loginTable.put(_students[i].getName(), _students[i].getLogin());
    }
    for (int i=0; i < _friends.length; i++) {
      if (loginTable.containsKey(_friends[i].getName())){
        String login = loginTable.get(_friends[i].getName());
        System.out.println(_friends[i] + "'s login is " + login + "!");
      }
    }
    ```
  - More efficient, $O(n)$ time complexity.
  - Note: If a key is not present, `get()` returns `null`.

**Finding out your friends’ logins (4/4)**
- An approach using a `HashMap`:
  - `public V put(K key, V value)`
  - `public V get(Object key)`
  - `public boolean containsKey(Object key)`
  - `public boolean containsValue(Object Value)`
  - `public V remove(Object key)`

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

  • New Approach:
    ```java
    HashMap<String, String> loginTable = new HashMap<String, String>();
    for (int i=0; i < _students.length; i++) {
      loginTable.put(_students[i].getName(), _students[i].getLogin());
    }
    for (int i=0; i < _friends.length; i++) {
      if (loginTable.containsKey(_friends[i].getName())){
        String login = loginTable.get(_friends[i].getName());
        System.out.println(_friends[i] + "'s login is " + login + "!");
      }
    }
    ```
  - More efficient, $O(n)$ time complexity.
  - Note: If a key is not present, `get()` returns `null`.
Finding out your friends’ logins (4/4)

- Using a HashMap

```java
HashMap<String, String> myTable = new HashMap<String, String>();
for (CSStudent student : _students) { //same array of students
    String login = student.getLogin(); //build HashMap
    myTable.put(student.getName(), login);
}

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 + "!");
}
```

- What’s the runtime now?
- O(n) – because each insert and search is O(1); much better!

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 Strings, each an individual word
  - String _searchTerm, the word you’re looking for

```java
int wordCounter = 0; //frequency of single term
for (String word : _book) {
    if (word.equals(_searchTerm)) {  //for each term, should we iterate through the _book for each of the search terms? Sounds like O(n^2)
        wordCounter++;
    }
}
System.out.println(_searchTerm + " appears " + wordCounter + " times");
```

Counting frequency in an Array (2/4)

```java
HashMap<String, Integer> countMap = new HashMap<String, Integer>();
//_book is an array of words
//_searchTerms is an array of Strings we’re counting
for (String word : _book) {
    if (countMap.containsKey(word)) {
        countMap.put(word, countMap.get(word) + 1);
    } else {
        countMap.put(word, 1);
    }
}
for (String word : _searchTerms) {
    Integer freq = countMap.get(word);
    if (freq == null) {
        freq = 0;
    }
    System.out.printf("%s shows up %d times\n", word, freq);
}
```

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?
  - if for each term, should we iterate through the _book for each of the search terms? Sounds like O(n^2)

```java
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
- Data structures we have so far:
  - runtime to search an unsorted array is O(n)
  - runtime to search a sorted array is O(log n)
  - can we do better this?
    - The next two lectures provide two different answers to this question
```
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

Map Implementation (3/4)
- But creating an array to look up CS15 students (value) based on some ID number (key) would be a tremendous waste of space
  - if ID number is one letter followed by five digits (e.g., D00011), there are $26 \times 10^5$ combinations!
  - (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 350 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 alphanumerics?
  - 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

Hash Functions (1/2)
- 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
Hash Functions (2/2)

- 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 them together, then % that result by table size
- 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!

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!

HashMap Pseudocode

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

public put(K key, V val):
    int index = hash(key)
    table[index].addFirst(key, val)
    O(1), if h() runs in O(1) time

public V get(K key):
    index = hash(key)
    for (k, v) in table[index]:
        if k == key:
            return v
    return null //key not found

Runs in O(k) time, where k is size of bucket, usually small
```

Note: LinkedLists only hold one element per node, so in actual code, you would need to make a class to hold the key and the 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
  - In the Trees lecture, we’ll show that there are more complex queries for which Hash Table is not optimal
- No support for ordering
  - (key, value) pairs are not stored in any logical order

Announcements

- Start working on Tetris!
  - early: 11/18
  - on time: 11/20
  - late: 11/22
- Data Structures and Algorithms mini-assignment out now!
- There will be a discussion next week covering topics learned in the last couple lectures
- Your last lab is this week!
- If you don’t finish during lab, you can get checked off during TA Hours
- Make sure to get checked off by end of hours on 11/15