CSE 143
Lecture 14

`AnagramSolver`

and

Hashing

slides created by Ethan Apter

http://www.cs.washington.edu/143/

Ada Lovelace (1815-1852)

- Ada Lovelace is considered the first computer programmer for her work on Charles Babbage's analytical engine

- She was a programmer back when computers were still theoretical!

<http://en.wikipedia.org/wiki/Ada_lovelace>

Alan Turing (1912-1954)

- Alan Turing made key contributions to artificial intelligence (the Turing test) and computability theory (the Turing machine)

- He also worked on breaking Enigma (a Nazi encryption machine)

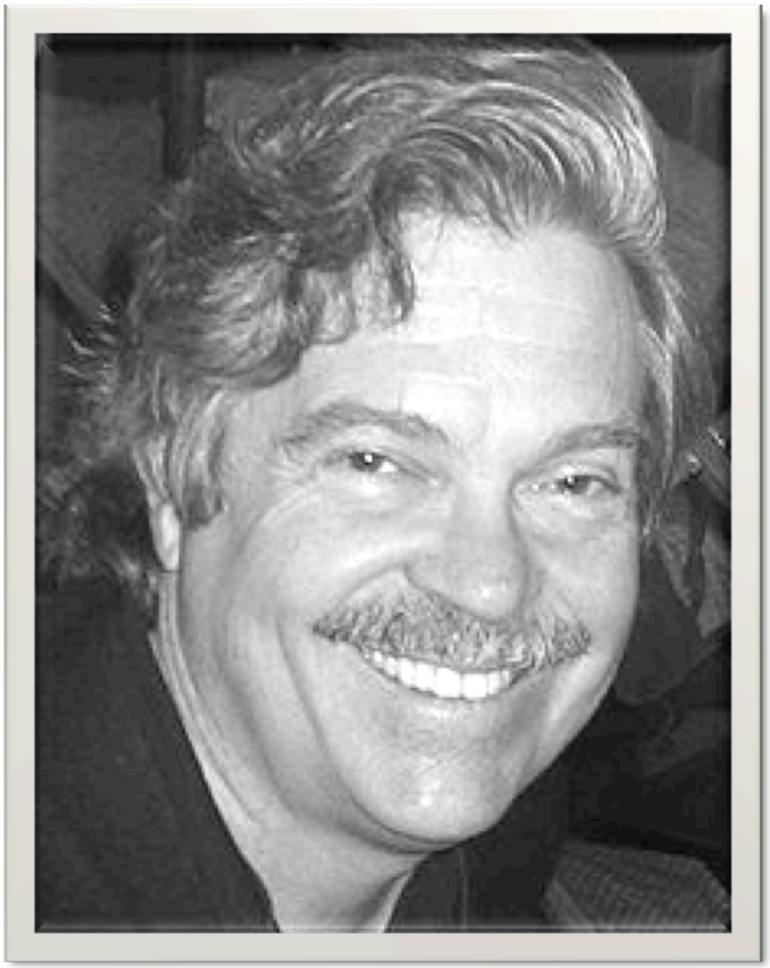
<http://en.wikipedia.org/wiki/Alan_turing>

Grace Hopper (1906-1992)

- Grace Hopper developed the first compiler
- She was responsible for the idea that programming code could look like English rather than machine code
- She influenced the languages COBOL and FORTRAN

<http://en.wikipedia.org/wiki/Grace_hopper>

Alan Kay (1940)



•Alan Kay worked on Object-Oriented Programming

•He designed SmallTalk, a programming language in which everything is an object

•He also worked on graphical user interfaces (GUIs)

<http://en.wikipedia.org/wiki/Alan_Kay>

John McCarthy (1927)

•John McCarthy designed Lisp ("Lisp" is short for "List Processing")

•He invented if/else

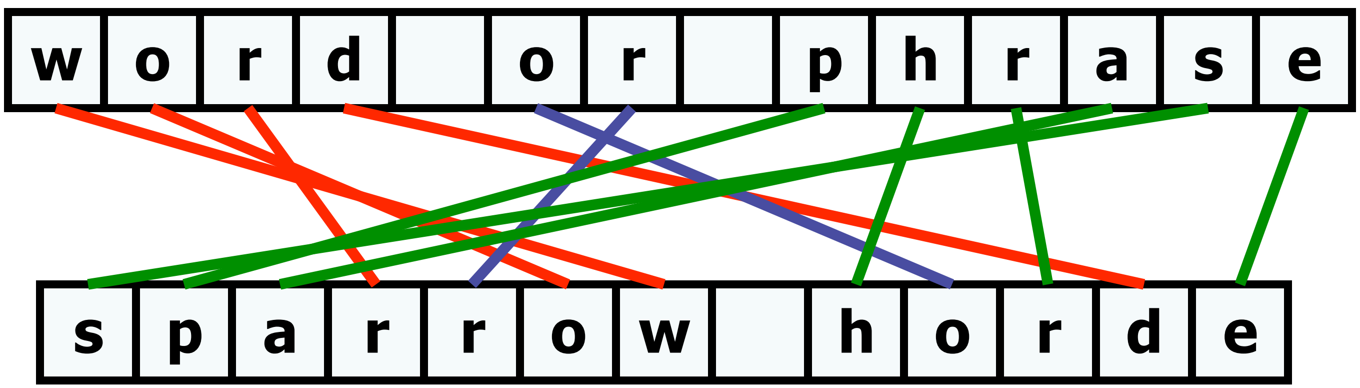
•Lisp is a very flexible language and was popular with the Artificial Intelligence community

<http://en.wikipedia.org/wiki/John_McCarthy_(computer_scientist)>
<http://en.wikipedia.org/wiki/Lisp_(programming_language)>
<http://www-formal.stanford.edu/jmc/jmcbw.jpg>

Anagrams

- **anagram:** a rearrangement of the letters from a word or phrase to form another word or phrase

- Consider the phrase "word or phrase"
  - one anagram of "word or phrase" is "sparrow horde"



- Some other anagrams:
  - "Alyssa Harding" → "darling sashay"
  - "Ethan Apter" → "ate panther"

AnagramSolver

- Your next assignment is to write a class named **`AnagramSolver`**

- **`AnagramSolver`** finds all the anagrams for a given word or phrase (within the specified dictionary)
  - it uses recursive backtracking to do this

- **`AnagramSolver`** may well be either the easiest or hardest assignment this quarter
  - easy: it's similar to 8 Queens, it's short (approx. 50 lines)
  - hard: it's your first recursive backtracking assignment

AnagramSolver

- Consider the phrase "Ada Lovelace"

- Some anagrams of "Ada Lovelace" are:
  - "ace dale oval"
  - "coda lava eel"
  - "lace lava ode"

- We could think of each anagram as a list of words:
  - "ace dale oval" → [ace, dale, oval]
  - "coda lava eel" → [coda, lava, eel]
  - "lace lava ode" → [lace, lava, ode]

AnagramSolver

- Consider also the small dictionary file dict1.txt:

ail	gnat	run
alga	lace	rung
angular	lain	tag
ant	lava	tail
coda	love	tan
eel	lunar	tang
gal	nag	tin
gala	natural	urinal
giant	nit	urn
gin	ruin	

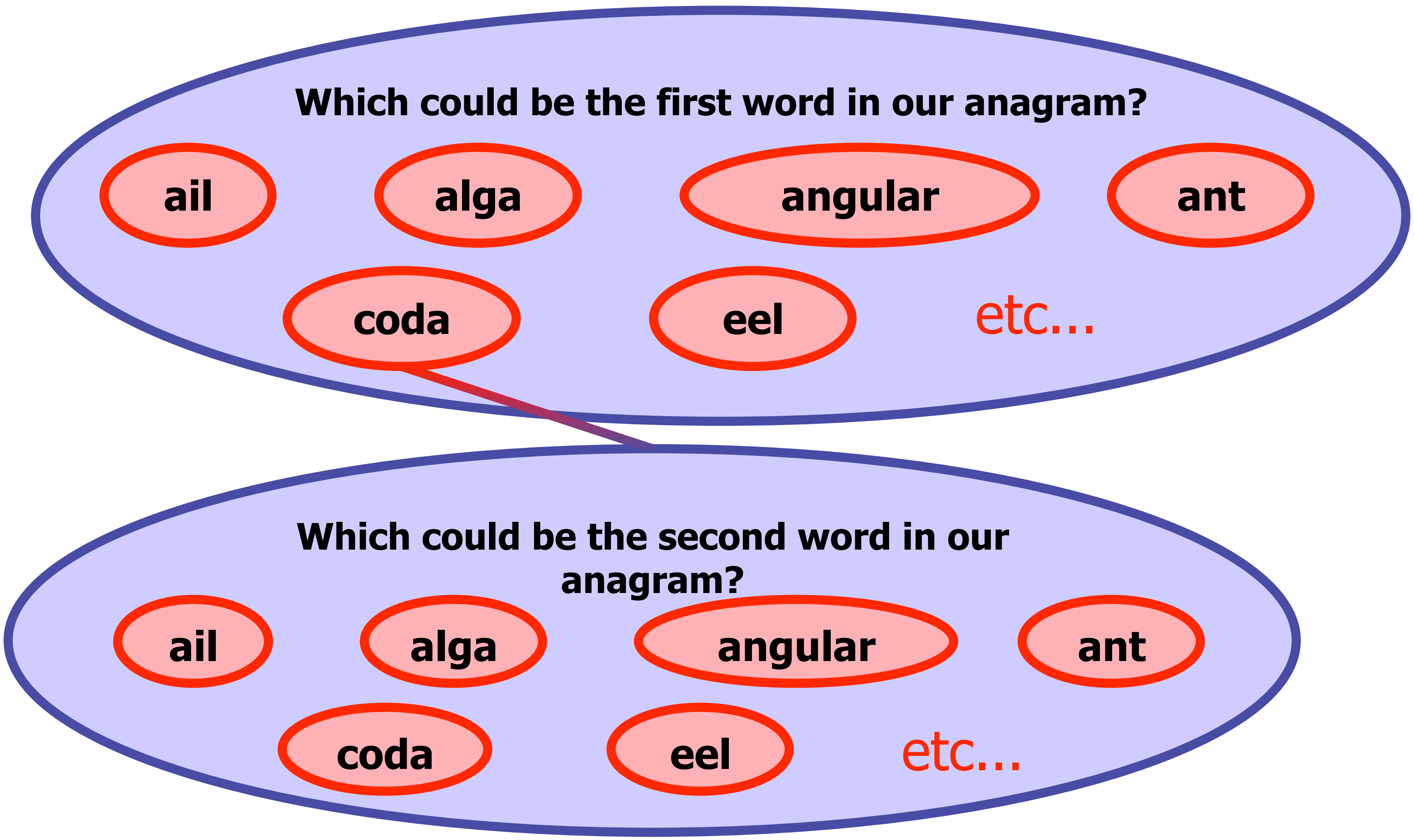
- We're going to use only the words from this dictionary to make anagrams of "Ada Lovelace"

AnagramSolver

- Which is the first word in this list that *could* be part of an anagram of "Ada Lovelace"
  - ail
    - no: "Ada Lovelace" doesn't contain an "i"
  - alga
    - no: "Ada Lovelace" doesn't contain a "g"
  - angular
    - no: "Ada Lovelace" doesn't contain an "n", a "g", a "u", or an "r"
  - ant
    - no: "Ada Lovelace" doesn't contain an "n" or a "t"
  - coda
    - yes: "Ada Lovelace" contains all the letters in "coda"

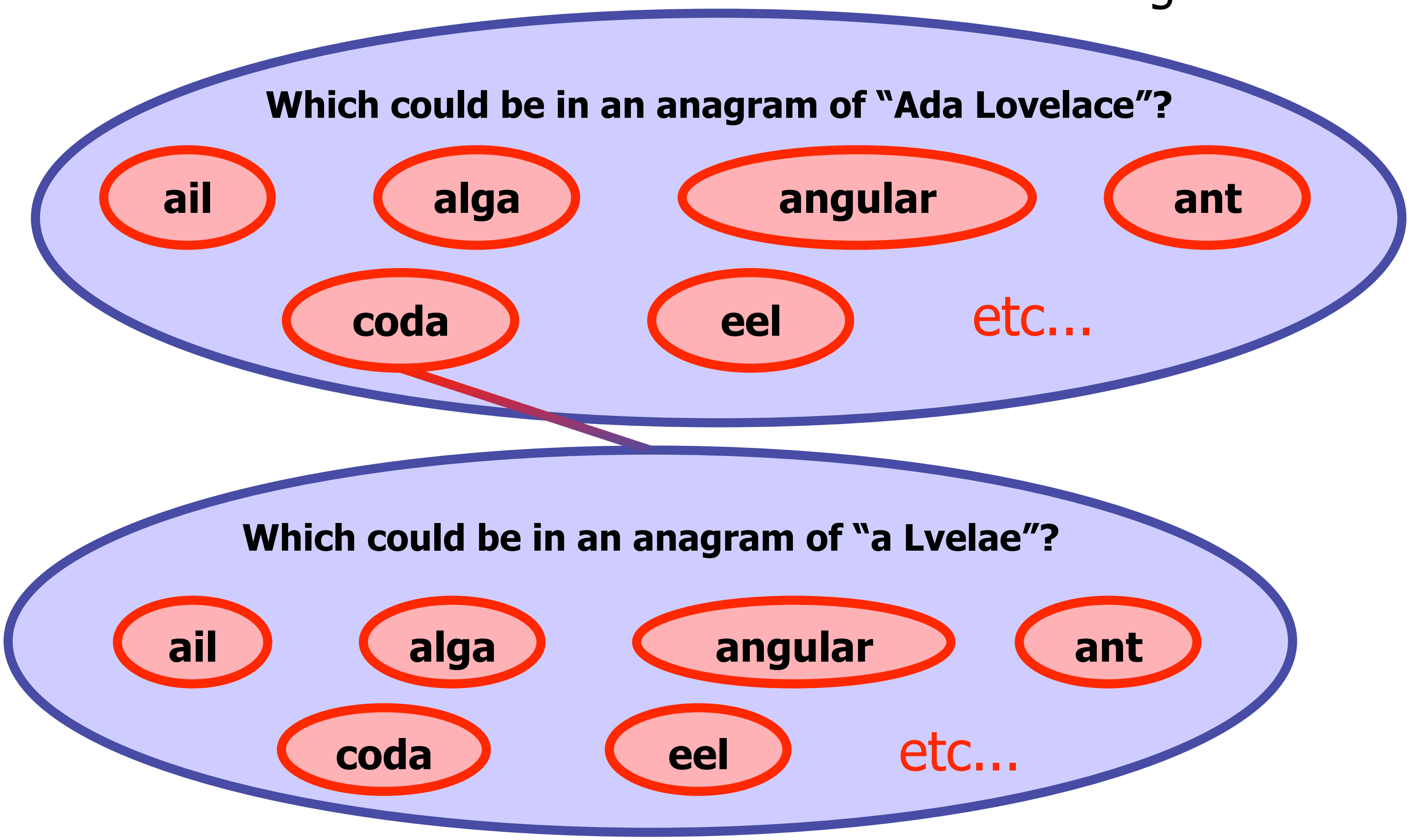
AnagramSolver

- This is just like making a choice in recursive backtracking:



AnagramSolver

- At each level, we go through all possible words
  - but the letters we have left to work with changes!



Low-Level Details

- Clearly there are some low level details here in deciding whether one phrase contains the same letters as another
- Just like 8 Queens had the Board class for its low-level details, we'll have a class that handles the low-level details of **`AnagramSolver`**
- This low-level detail class is called **`LetterInventory`**
  - as you might have guessed, it keeps track of letters
- And we'll give it to you!

LetterInventory

- **LetterInventory** has the following methods (described further in the write-up):

```
public LetterInventory(String s)

public void add(LetterInventory li)

public boolean contains(LetterInventory li)

public boolean isEmpty()

public int size()

public void subtract(LetterInventory li)

public String toString()
```

LetterInventory

- Let’s construct and print a **`LetterInventory`**:

```
LetterInventory li = new LetterInventory(“Hello”);
li.isEmpty();                 // returns false
li.size();                    // returns 5
System.out.println(li);       // prints [ehllo]
```

- **`li`** contains 1 e, 1 h, 2 l’s, and 1 o

- We can also do some operations on **`li`**:

```
LetterInventory li2 = new LetterInventory(“heel”);
li.contains(li2);             // returns false
li.add(li2);
System.out.println(li);       // prints [eeehhlllo]
li.contains(li2);             // returns true
li.substract(li2);
System.out.println(li);       // prints [ehllo]
```

AnagramSolver

- AnagramSolver has a lot in common with 8 Queens
  - I can't stress this enough! If you understand 8 Queens, writing AnagramSolver shouldn't be too hard

- Key questions to ask yourself on this assignment:
  - When am I done?
    - for 8 Queens, we were done when we reached column 9
  - If I'm not done, what are my options?
    - for 8 Queens, the options were the possible rows for this column
  - How do I make and un-make choices?
    - for 8 Queens, this was placing and removing queens

AnagramSolver

- You must include two optimizations in your assignment
  - because backtracking is inefficient, we need to gain some speed where we can

- You must preprocess the dictionary into **LetterInventory**s
  - you'll store these in a Map
    - specifically, in a HashMap, which is slightly faster than a TreeMap

- You must prune the dictionary before starting the recursion
  - by "prune," we mean remove all the words that couldn't possibly be in an anagram of the given phrase
  - you need do this only once (before starting the recursion)

Maps

- Recall that Maps have the following methods:

```
// adds a mapping from the given key to the given value
void put(K key, V value)

// returns the value mapped to the given key (null if none)
V get(K key)

// returns true if the map contains a mapping for the given key
boolean containsKey(K key)

// removes any existing mapping for the given key
remove(K key)
```

- A HashMap can perform all of these operations in O(1)
  - that's really fast!
  - this makes HashMaps really useful for many applications

Hashing

- In order to do these operations quickly, HashMaps don't attempt to preserve the order of their keys and values

- Consider the following `int` array with 4 valid values:

0	1	2	3	4	5	6	7	8	9
3	7	11	26	0	0	0	0	0	0

- What would be a better order for fast access?

0	1	2	3	4	5	6	7	8	9
0	11	0	3	0	0	26	7	0	0

Hashing

- **hashing:** mapping a value to an integer index
- **hash table:** an array that stores elements by hashing
- **hash function:** an algorithm that maps values to indexes
  - e.g. hashFunction(value) → Math.abs(value) % arrayLength

```
11 % 10 == 1      (11 inserted at index 1)
 3 % 10 == 3       (3 inserted at index 3)
26 % 10 == 6      (26 inserted at index 6)
 7 % 10 == 7       (7 inserted at index 7)
```

0	1	2	3	4	5	6	7	8	9
0	11	0	3	0	0	26	7	0	0

Hashing

- So far, we've treated keys and values like they're the same thing, but they're not
  - the key is used to located and identify the value
  - the value is the information that we want to store/retrieve

- With maps, we work with both a key and a value
  - we hash the key to determine the index
  - ...and then we store the value at this index

- So what we've done so far is:
  - with a key of 11, add the value 11 to the array
  - with a key of 3, add the value 3 to the array
  - etc

Hashing

- But we don't have to make the key the same as the value

- Consider the array from before:

0	1	2	3	4	5	6	7	8	9
0	11	0	3	0	0	26	7	0	0

- This is what happens if we use a key of 8 to add value 4:

0	1	2	3	4	5	6	7	8	9
0	11	0	3	0	0	26	7	4	0

- But notice that our key (8) is completely gone

Hashing

- Now we can support all the simple operations of a Map:
  - put(key, value)
    ```
    int index = hashFunction(key);
    array[index] = value;
    ```
  - get(key)
    ```
    return array[hashFunction(key)];
    ```
  - remove(key)
    ```
    array[hashFunction(key)] = 0;
    ```
- But what happens if another value is already there?

Collisions

- If we use a key of 41 to add value 5 to our array, we'll overwrite the old value (11) at index 1:

0	1	2	3	4	5	6	7	8	9
0	5	0	3	0	0	26	7	4	0

- This is called a collision

- **collision:** when a hash functions maps more than one element to the same index
  - collisions are bad
  - they also happen a lot

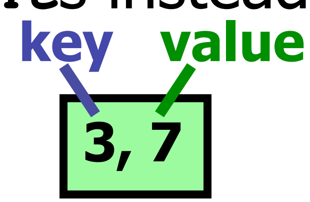
- **collision resolution:** an algorithm for handling collisions

Collisions

- To handle collisions, we first have to be able to tell the keys and values apart
  - we've been remembering the values
  - but we also need to remember the original key!
- Consider the following simple class:

```
public class IntInt {
    public int key;
    public int value;
}
```

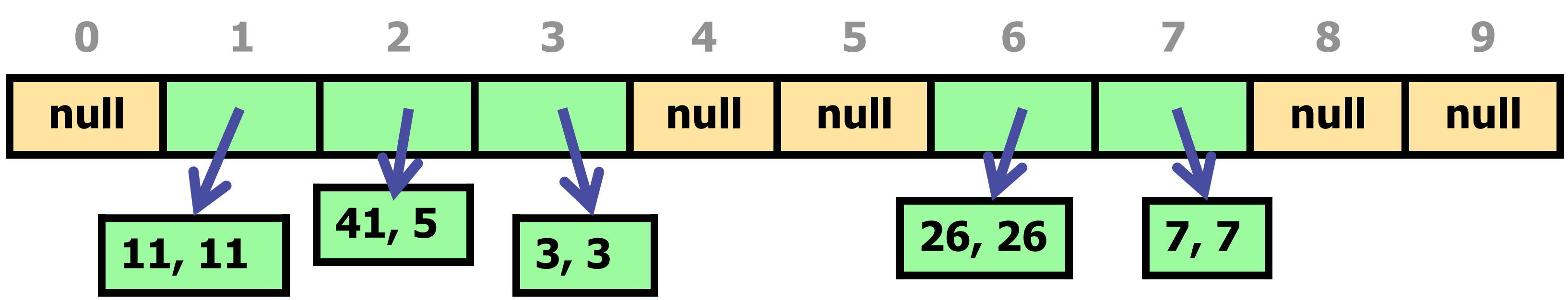
- We'll make an array of `IntInts` instead of regular `ints`
- I'll draw `IntInts` like this:



Probing

- **probing:** resolving a collision by moving to another index
  - **linear probing:** probes by moving to the *next* index

```
// put(key, value)
put(11, 11)
put(3, 3)
put(26, 26)
put(7, 7)
put(41, 5)    // bumped to index 2 instead
```



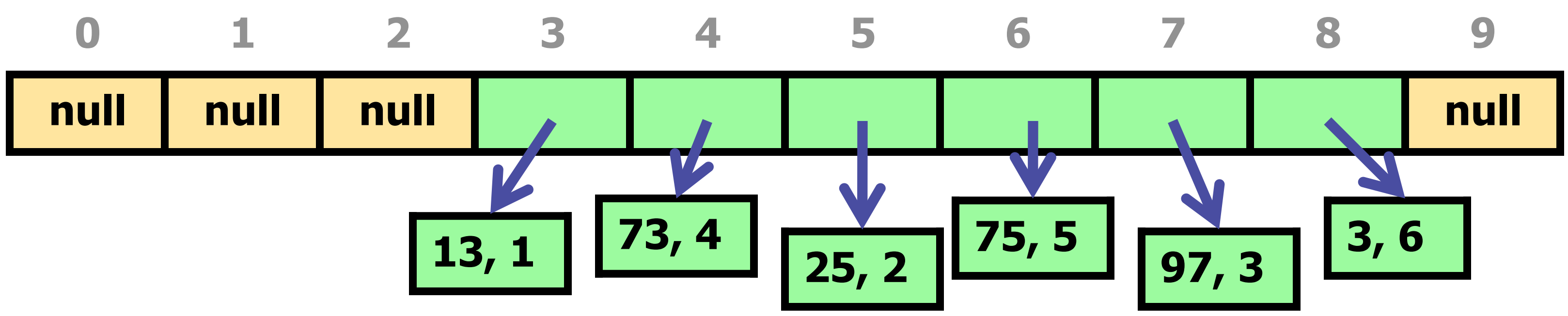
- If we look at the keys, we can still tell if we've found the right object (even if it's not where we first expect)

Clustering

- Linear probing can lead to clustering

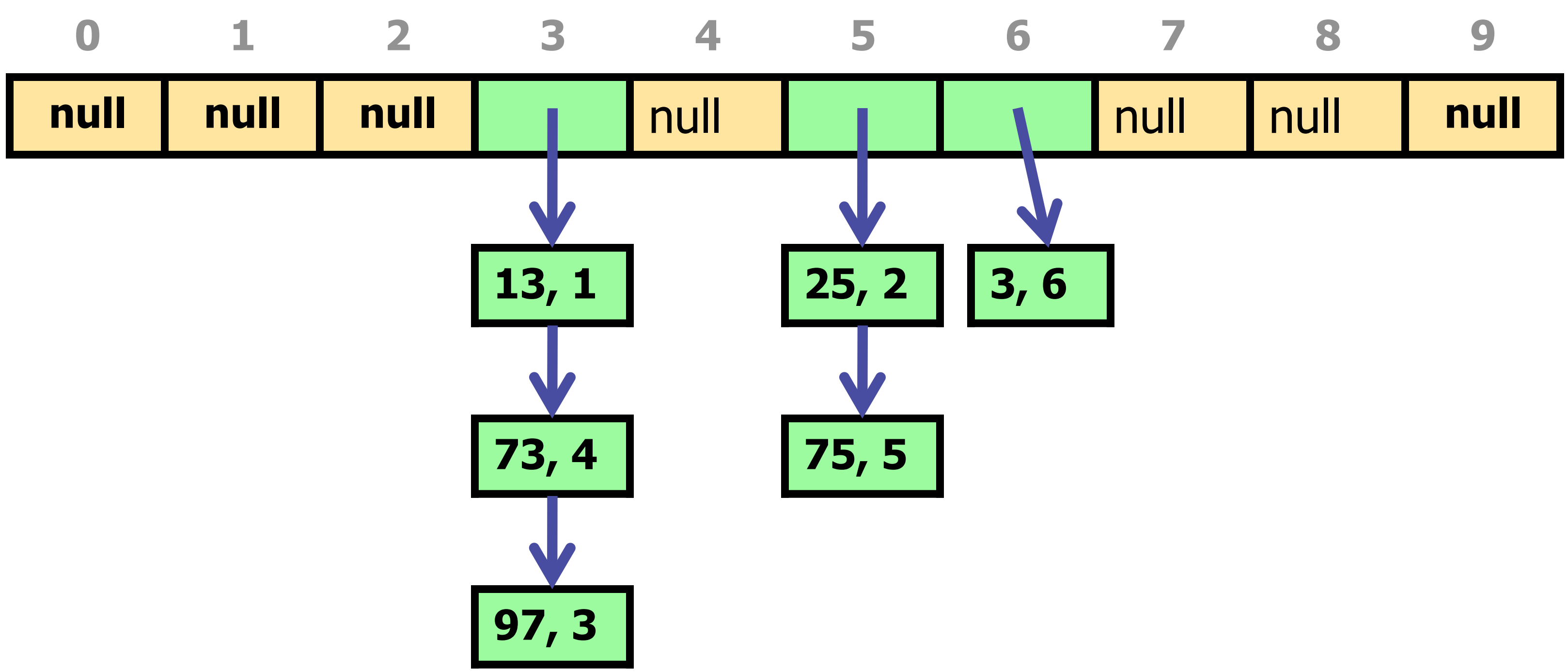
- **clustering:** groups of elements at neighboring indexes
  - slows down hash table lookup (must loop over elements)

```
put(13, 1)
put(25, 2)
put(97, 3)
put(73, 4)   // collides with 1
put(75, 5)   // collides with 2
put(3, 6)    // collides with 1, 4, 2, 5, and 3!
```



Chaining

- **chaining:** resolving collisions by storing a list at each index
  - we still must traverse the lists
  - but ideally the lists are short
  - and we never run out of room



Rehashing

- **rehash:** grow to larger array when table becomes too full
  - because we want to keep our O(1) operations
  - we can't simply copy the old array to the new one. Why?

- If we just copied the old array to the new one, we might not be putting the keys/values at the right indexes
  - recall that our hash function uses the array length
  - when the array length changes, the result from the hash function will change, even though the keys are the same
  - so we have to rehash every element

- **load factor:** ratio of (# of elements) / (array length)
  - many hash tables grow when load factor ≈ 0.75

Hashing Objects

- It's easy to hash **`ints`**
  - but how can we hash non-**`ints`**, like objects?
- We'd have to convert them to **`ints`** somehow
  - because arrays only use **`ints`** for indexes
- Fortunately, Object has the following method defined:

```
// returns an integer hash code for this object
public int hashCode()
```

- The implementation of **`hashCode()`** depends on the object, because each object has different data inside
  - String's **`hashCode()`** adds the ASCII values of its letters
  - You can also write a **`hashCode()`** for your own Objects