Announcements

Midterm 2:

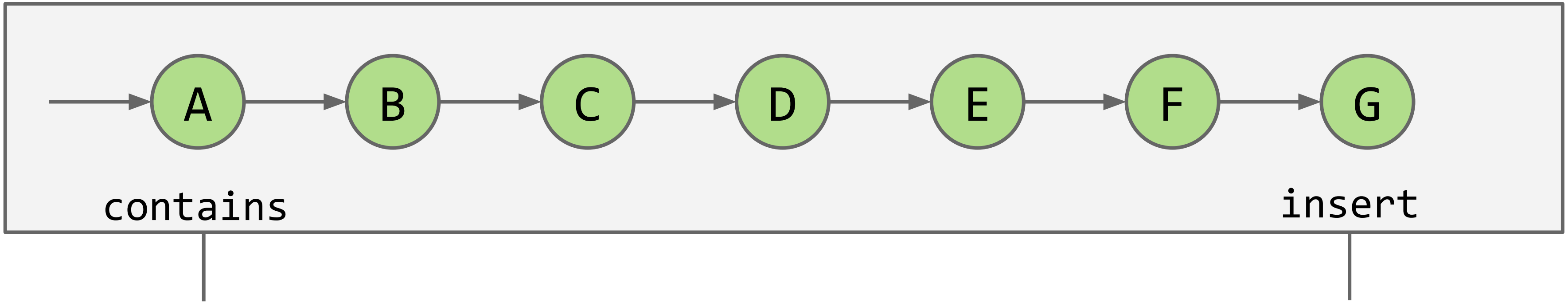
- 3/20, 8-10 PM in various rooms.
- Covers material through 3/16 (next Friday).
- Study using study guides.
  - THE KEY IS METACOGNITION: Reflect on your problem solving strategies and those of your fellow students.
  - Understanding a handful of solutions to old midterm problems is less helpful than you might think -- look at answers as late as possible.
- There is an alternate 61C midterm from 6 - 8 in 1 LeConte.

CS61B

Lecture 23: Hashing

- Set Implementations, DataIndexedIntegerSet
- Binary Representations, DataIndexedSet
- Handling Collisions
- Hash Functions

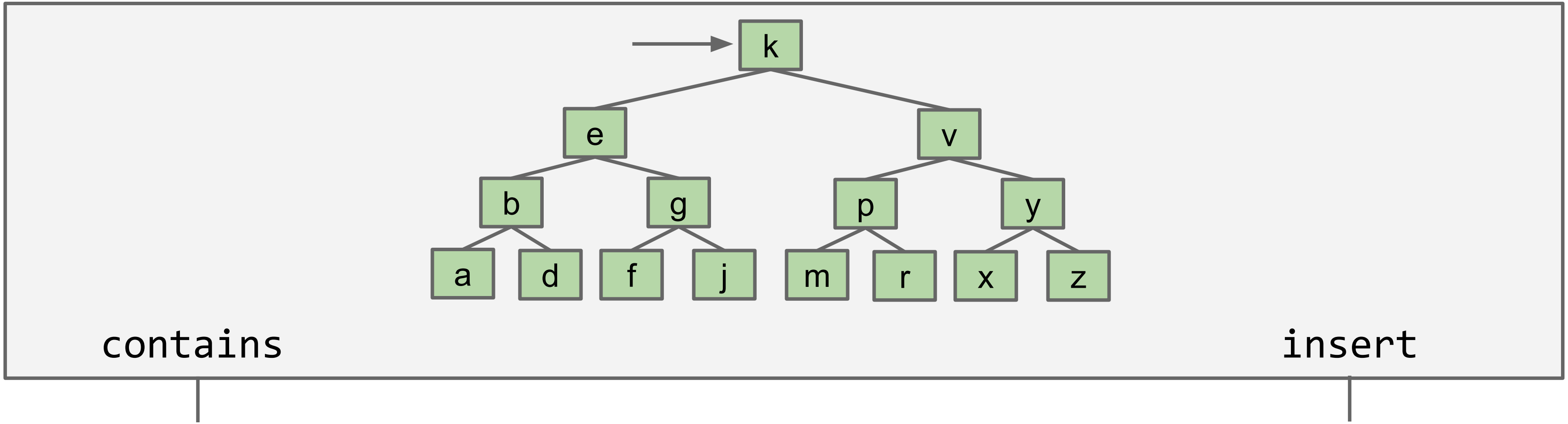
Techniques for Storing Data: Ordered Linked List



	contains(x)	insert(x)
Linked List	Θ(N)	Θ(N)

Worst case runtimes

Techniques for Storing Data: Bushy BST



Limitations:

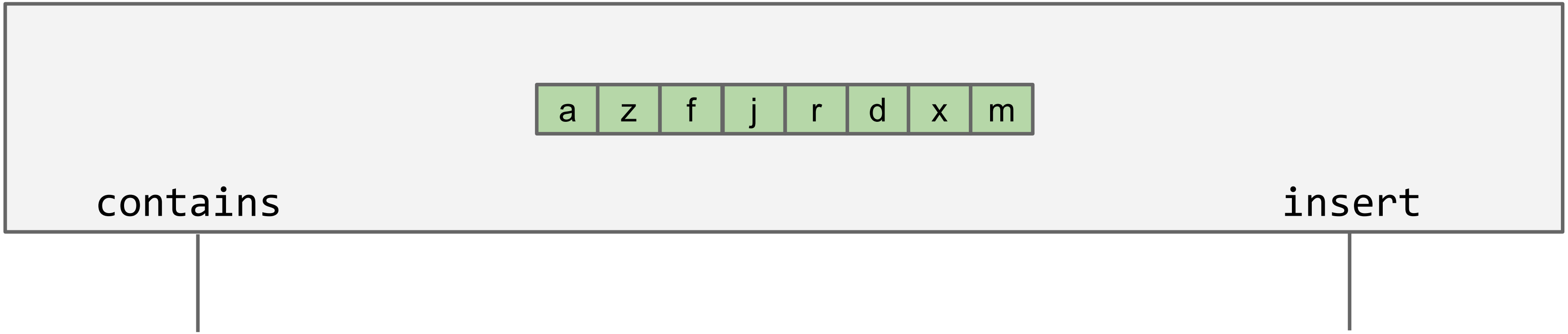
- Items must be comparable.
- Maintaining bushiness is non-trivial.
- Θ(log N), can we do better?

	contains(x)	insert(x)
Linked List	Θ(N)	Θ(N)
Bushy BSTs	Θ(log N)	Θ(log N)

Worst case runtimes

Note: log is pretty good. One billion items yields tree height of only 30.

Techniques for Storing Data: Unordered Array



	contains(x)	insert(x)
Linked List	Θ(N)	Θ(N)
Bushy BSTs	Θ(log N)	Θ(log N)
Unordered Array	Θ(N)	Θ(N)

Worst case runtimes

Unordered arrays are terrible (and so are ordered ones).

But a third type of array...

Using data as an Index

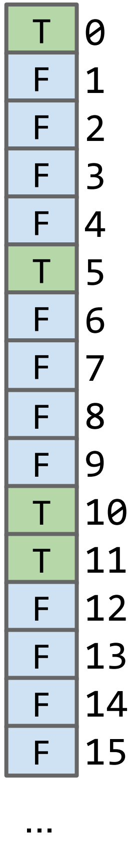
One extreme approach: All data is really just bits.

- Use data itself as an array index.
- Store true and false in the array.

Downsides of this approach (that we can maybe fix):

- Extremely wasteful of memory. To support checking presence of all positive integers, we need 2 billion booleans.
- Need some way to generalize beyond integers.

```
DataIndexedIntegerSet diis = new DataIndexedIntegerSet();
diis.insert(0);
diis.insert(5);
diis.insert(10);
diis.insert(11);
```



Set containing 0, 5, 10, 11

DataIndexedIntegerSet Implementation

```java
public class DataIndexedIntegerSet {
    boolean[] present;

    public DataIndexedIntegerSet() {
        present = new boolean[16];
    }

    public insert(int i) {
        present[i] = true;
    }

    public contains(int i) {
        return present[i];
    }
}
```

T	0
F	1
F	2
F	3
F	4
T	5
F	6
F	7
F	8
F	9
T	10
T	11
F	12
F	13
F	14
F	15

...

Set containing 0, 5, 10, 11

DataIndexedIntegerSet Implementation

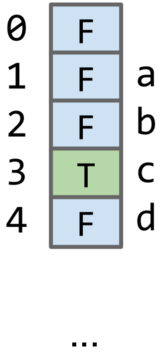
```
public class DataIndexedIntegerSet {
    boolean[] present;

    public DataIndexedIntegerSet() {
        present = new boolean[100000];
    }

    public insert(int i) {
        present[i] = true;
    }

    public contains(int i) {
        return present[i];
    }
}
```

	contains(x)	insert(x)
Linked List	Θ(N)	Θ(N)
Bushy BSTs	Θ(log N)	Θ(log N)
Unordered Array	Θ(N)	Θ(N)
DataIndexedArray	Θ(1)	Θ(1)

Worst case runtimes

Binary Representations
DataIndexedSet

Generalizing the DataIndexedIntegerSet Idea



Suppose we want to insert(“cat”)

The key question:

- What is the `cat`th element of an array?
- One idea: Use the first letter of the word as an index.

What’s wrong with this approach?

- 0 never changes (so a tiny bit of wasted space).
- Other words start with c.
    - contains(“chupacabra”) : YES
- Can’t store “=98yae98fwyawef”

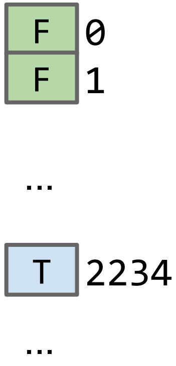
Refined Approach

Treat the string as a n-digit base 27 number.

- c: 3rd letter of alphabet, a: 1st letter, t: 20th letter
- Thus the index of “cat” is $3 * 27^2 + 1 * 27 + 20$

Why this specific pattern?

- Let’s review how numbers are represented in decimal.



The Decimal Number System

In the decimal number system, we have 10 digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Want numbers larger than 9? Use a sequence of digits.

Example: 7091 in base 10

- $7091_{10} = (7 \times 10^{3}) + (0 \times 10^{2}) + (9 \times 10^{1}) + (1 \times 10^{0})$

Binary (Base 2)

In the binary number system, we have two digits: 0, 1.

Want larger numbers than 1? Use a sequence of digits.

Example: What is 1110_2 in base 10?

- $1110_2 = (1 \times 2^3) + (1 \times 2^2) + (1 \times 2^1) + (0 \times 2^0)$
  $= (8) + (4) + (2) + (0)$
  $= 14_{10}$

Base 10	Base 8	Base 2
0	0	0
1	1	1
2	2	10
3	3	11
4	4	100
5	5	101
6	6	110
7	7	111
8	10	1000
9	11	1001

Binary (Base 2): Larger Example

Suppose we have the 32 bit binary number below.

- What is it decimal? Sum the 2nd, 4th, 5th, 10th, and 11th powers of 2.

```
00 00000 00000 00000 00011 00001 10100
```

2^{11} 2^{10} 2^{5} 2^{4} 2^{2}

- $(1 \times 2^{\mathbf{11}}) + (1 \times 2^{\mathbf{10}}) + (1 \times 2^{\mathbf{5}}) + (1 \times 2^{\mathbf{4}}) + (1 \times 2^{\mathbf{2}})$

  $= (2048) + (1024) + (32) + (16) + (4)$

  $= 3124_{10}$

Generalizing to Words

There are many ways to represent cat.

Base 27: Our approach before: cat ⟶ $3 * 27^2 + 1 * 27 + 20 = 2234$

`00 00000 00000 00000 00010 00101 11010` 2234

F	0
F	1
...	
T	2234

Base 32: $3 * 32^2 + 1 * 32 + 20 = 3124$

Why does this work?
Multiplying by 32 is equivalent
to shifting right by 5 places.

- Nice feature: Each letter is exactly 5 bits. Good for lecture.

`00 00000 00000 00000 00011 00001 10100` 3124

$\underbrace{00011}_{\text{c} \; 3} \; \underbrace{00001}_{\text{a} \; 1} \; \underbrace{10100}_{\text{t} \; 20}$

F	0
F	1
...	
T	3124

Generalizing to Words

Suppose we're storing multiple words.

- Convert each word to unique integer representation.
- As on previous slide: Use 5 bits per letter.
  - example: "cat" becomes 3124.
  - c: 3rd letter of alphabet, a: 1st, t: 20th

Equivalent to treating like a base 32 number.

$$\texttt{00 00000 00000 00000 }\underbrace{\texttt{00011}}_{c}\ \underbrace{\texttt{00001}}_{a}\ \underbrace{\texttt{10100}}_{t}$$

```
DataIndexedWordSet diws = new DataIndexedWordSet();
diws.put("cat");      ——→ 3124
diws.put("dog");      ——→ 4583
diws.put("potato");   ——→ 553256591
diws.put("snack");    ——→ 20382827
```

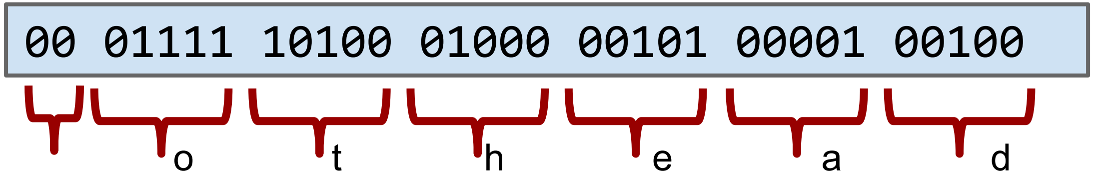
F	0
F	1
…	
T	3124
…	
T	4583
…	
T	20382827
…	
F	524555300
…	
T	553256591
F	553256592
…	

Generalizing to Words

What about longer strings? Have to tolerate either:

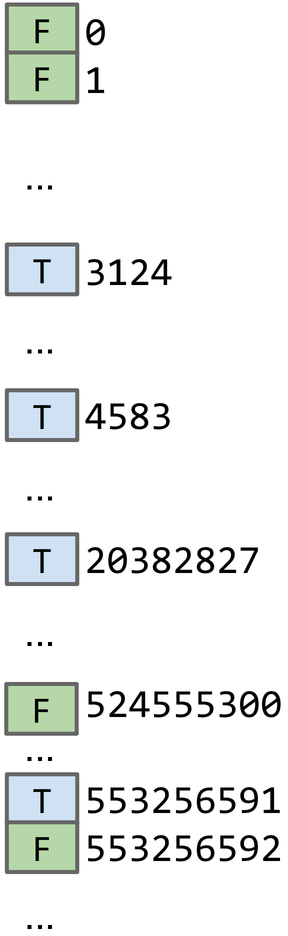
- A maximum string (seems lame).
- Ambiguity: e.g. “hothead” vs. “pothead”.
    - Both represented by 523878693

Switching to base 27 from 32 won’t save us (more soon).



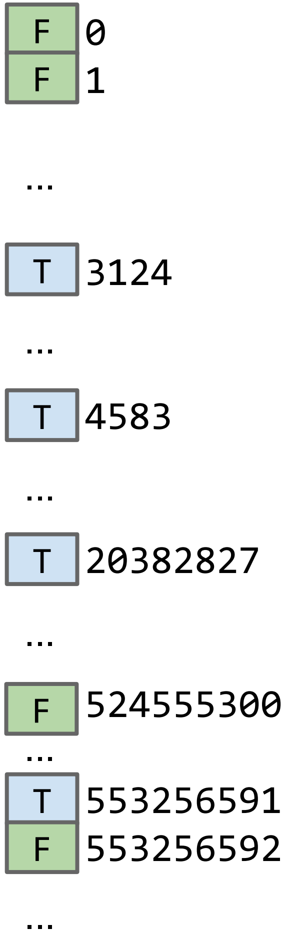
bottom bits of p
bottom bits of h

hothead?
pothead?
othead?



DataIndexedWordSet Implementation

```java
public void insert(String s) {
    int intRep = convertToInt(s);
    present[intRep] = true;
}

public boolean contains(String s) {
    int intRep = convertToInt(s);
    return present[intRep];
}
```



DataIndexedWordSet Implementation

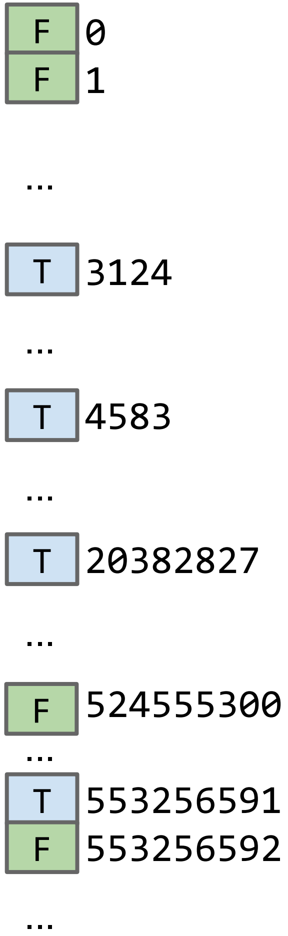
```java
/** Converts ith character of String to a letter number.
  * e.g. 'a' -> 1, 'b' -> 2, 'z' -> 26 */
public static int letterNum(String s, int i) {
    int ithChar = s.charAt(i);
    if ((ithChar < 'a') || (ithChar > 'z'))
        { throw new IllegalArgumentException();
    return ithChar - 'a' + 1;
}

public static int convertToInt(String s) {
    int intRep = 0;
    for (int i = 0; i < s.length(); i++) {
        intRep = intRep << 5; // same as intRep * 32;
        intRep = intRep + letterNum(s, i);
    }
    return intRep;
}
```

F 0
F 1
...
T 3124
...
T 4583
...
T 20382827
...
F 524555300
...
T 553256591
F 553256592
...

DataIndexedArray

Two fundamental challenges:

- How do we resolve ambiguity (“grosspie” vs. “bosspie”)?
  - We’ll call this ***collision handling***.
- How do we convert arbitrary data to an index?
  - We’ll call this ***computing a hashCode***.
  - For Strings, this was relatively straightforward (treat as a base 27 or base 32 number).
  - Note: Java requires that EVERY object provide a method that converts itself into an integer: hashCode()
  - More on what makes a good hashCode() later.



Hat Giveaway

RAILWORKS®

North America's leader in Track and Transit & Systems construction and maintenance services

Handling Collisions

Resolving Ambiguity

Biggest array in Java is 2 billion entries.

- Pigeonhole principle tells us that if there are more than 2 billion possible items, multiple items will share the same box.
- Example: More than 2 billion possible Planets.
  - Each has mass, xPos, yPos, xVel, yVel, imgName.
- Example: More than 2 billion possible strings.
  - “one”, “two”, … “four billion and six”, ...

Resolving Ambiguity

Pigeonhole principle tells us that if there are more than 2 billion possible things, multiple items will share the same box.

Suppose N items have the same hashcode h:

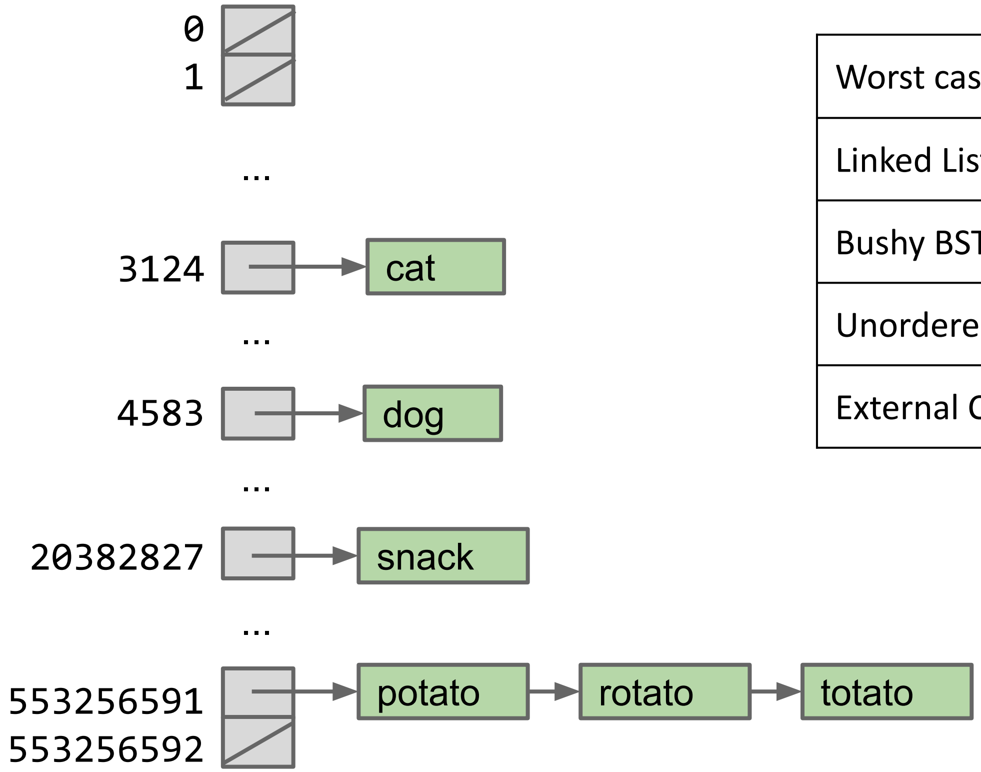
- Instead of storing `true` in position h, store a list of these N items at position h.

How to implement list?

- Easiest way: Linked list.
- But any list would do (ArrayList, etc.)
- (... if you wanted, could use a set instead)

External Chaining

External Chaining: Storing all items that map to h in a linked list.



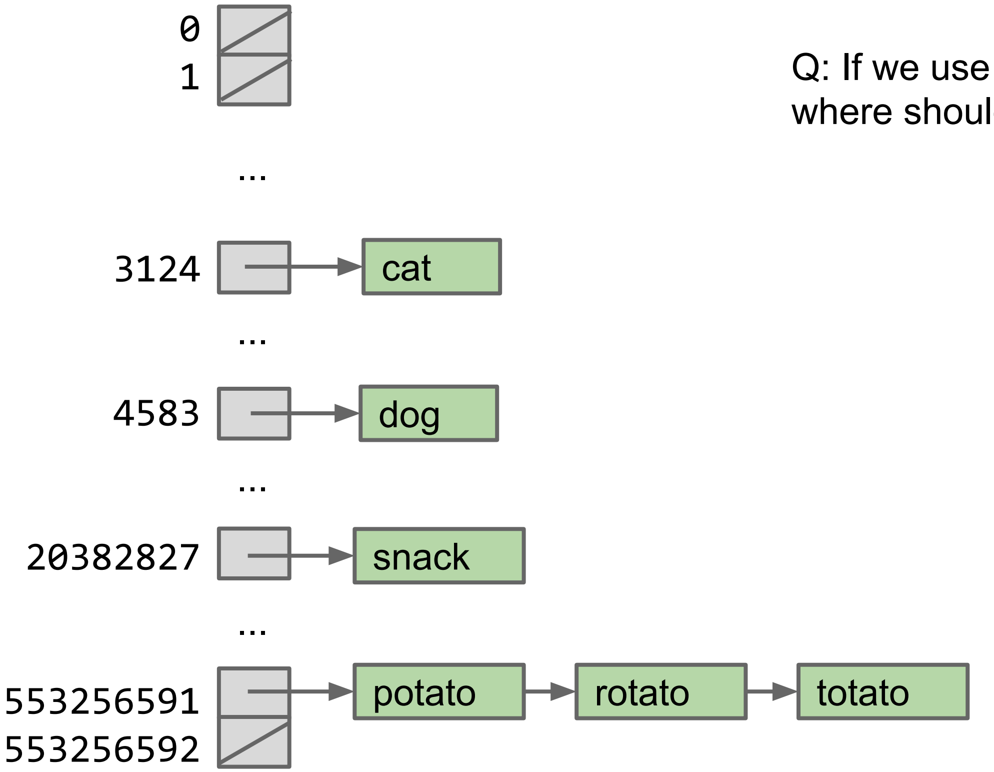
Worst case time	contains(x)	insert(x)
Linked List	Θ(N)	Θ(N)
Bushy BSTs	Θ(log N)	Θ(log N)
Unordered Array	Θ(N)	Θ(N)
External Chaining	Θ(Q)	Θ(Q)

Why Q and not 1?

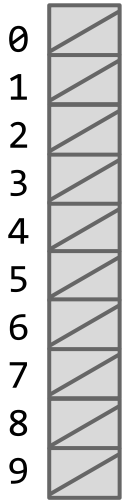
Q: Length of longest list

External Chaining

Observation: We don't really need 2 billion buckets.

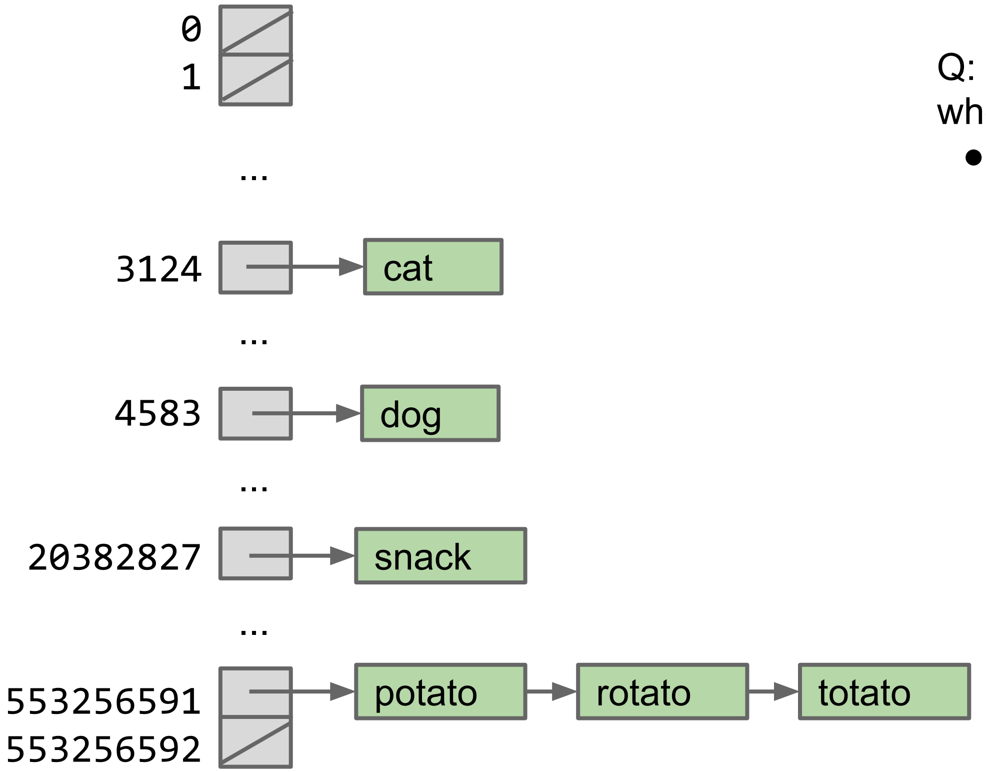


Q: If we use the 10 buckets on the right, where should our six items go?



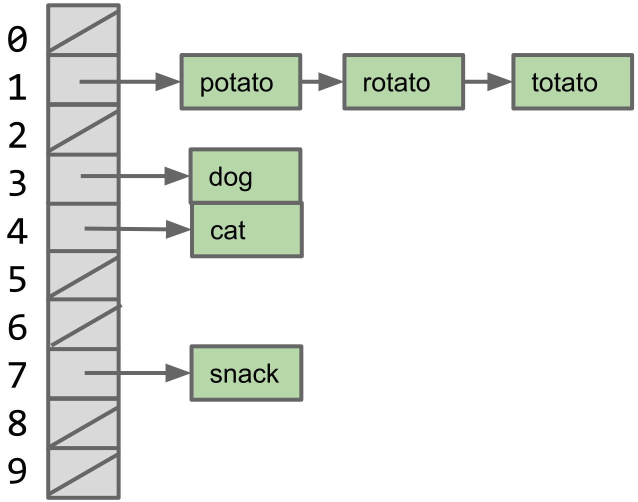
External Chaining

Observation: Can use modulus of hashcode to reduce bucket count.



Q: If we use the 10 buckets on the right, where should our six items go?

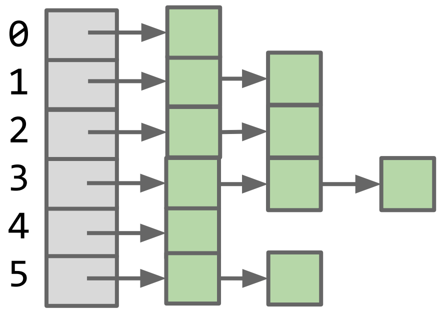
- Put in bucket = hashCode % 10



External Chaining Performance

Depends on the number of items in the 'bucket'.

- If N items are distributed across M buckets, average time grows with N/M = L, also known as the ***load factor***.
  - Average runtime is Θ(L).



Load factor: 11/6 = 1.83

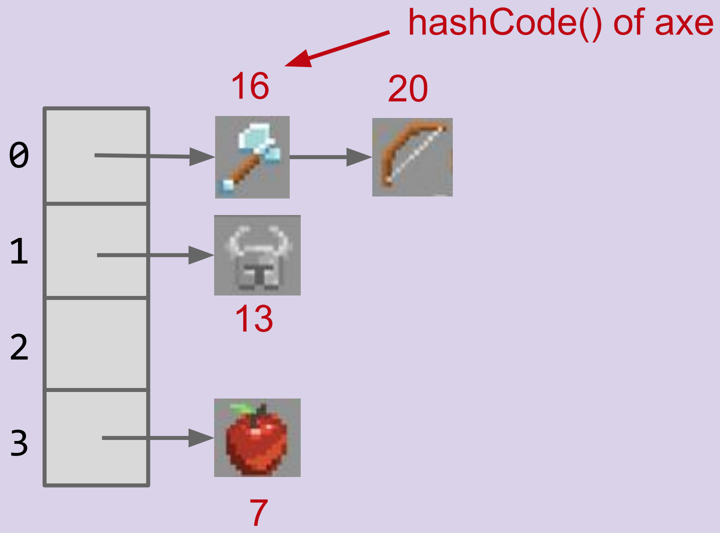
Obvious observation: If L is small, our data structure will be very fast.

Question: As N grows, what can we do to ensure that L stays small?

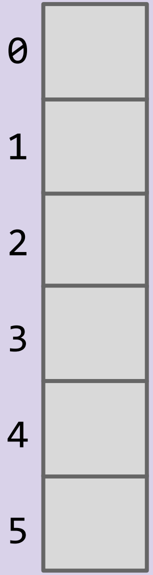
Array Resizing: http://yellkey.com/skin

Whenever L=N/M exceeds some number, increase M by resizing.

- Question: In which bin will the apple appear after resizing?



Load factor: 4/4 = 1

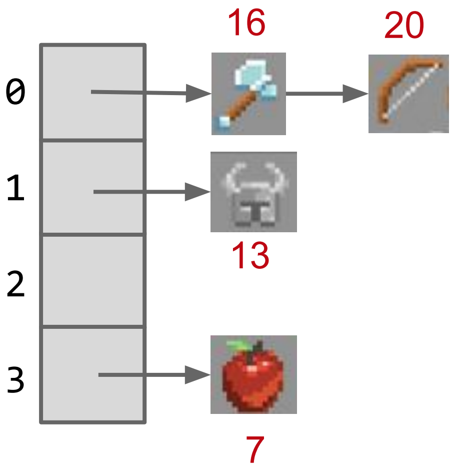


Load factor: 4/6 = 0.667

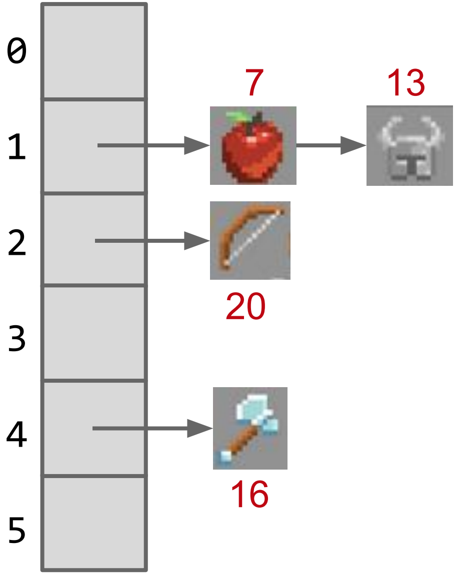
Array Resizing

Whenever L=N/M exceeds some number, increase M by resizing.

- Question: In which bin will the apple appear after resizing?



Load factor: 4/4 = 1



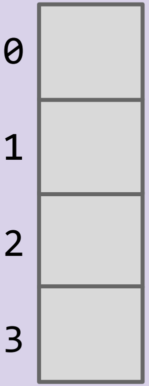
Load factor: 4/6 = 0.667
7 % 6 = 1

Using Negative .hashCodes: http://yellkey.com/medical

Suppose that 

`.hashCode()` returns `-1`.

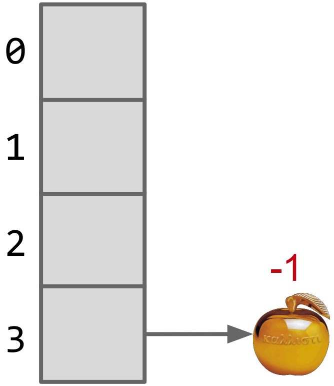
- Philosophically, into which bucket is it most natural to place this item?



Using Negative .hashCodes

Suppose that  `.hashCode()` returns `-1`.

- Philosophically, into which bucket is it most natural to place this item?
  - I say 3, since -1 → 3, 0 → 0, 1 → 1, 2 → 2, 3 → 3, 4 → 0, ...

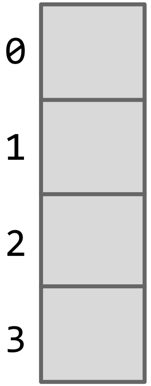


Using Negative .hashCodes in Java

Suppose that 

.hashCode() returns -1.

- Unfortunately, -1 % 4 = -1. Will result in index errors!
- Use Math.floorMod instead.




```java
public class ModTest {
  public static void main(String[] args) {
    System.out.println(-1 % 4);
    System.out.println(Math.floorMod(-1, 4));
  }
}
```

```
$ java ModTest
-1
3
```

Hash Table Definition and Key Implementation Details

This data structure we've designed is called a ***hash table***:

- Every item is mapped to a bucket number using a hash function.
- Typically, computing hash function consists of two steps:
  1. Computing a hashCode (integer between -2^{31} and $2^{31} - 1$).
  2. Computing index = hashCode modulo M.
- If L = N/M gets too large, increase M.

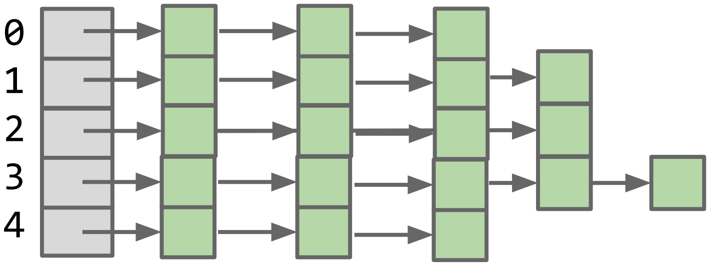
Be careful, negative numbers % M won't work.

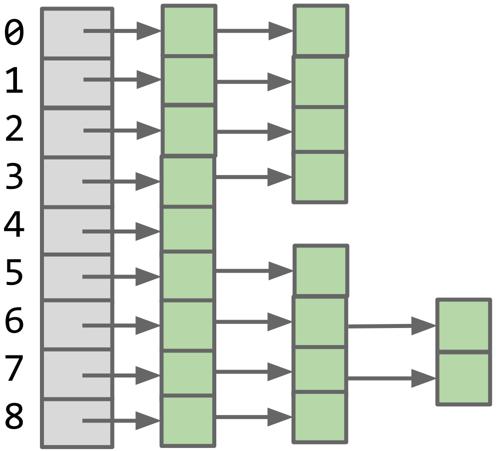
If multiple items map to the same bucket, we have to resolve ambiguity somehow. Two common techniques:

- External Chaining (creating a list for each bucket, the technique we just used).
- Open Addressing (a little stranger, not necessarily better, see extra slides).
  - May come up at job interviews.

External Chaining Performance

Assuming items are spread out (e.g. not all in the same bucket):





Average case time	contains(x)	insert(x)
External Chaining, Fixed Size	Θ(L)	Θ(L)
External Chaining With Resizing	Θ(L)	Θ(L)
Balanced BST	Θ(log N)	Θ(log N)

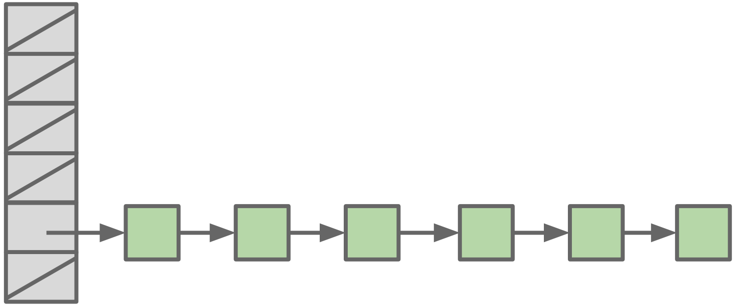
	Load Factor L
External Chaining, Fixed Size	Θ(N)
External Chaining With Resizing	Θ(1)

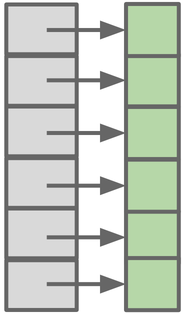
Amortized!

One Last Little Detail

Performance depends on the number of items in each ‘bucket’.

- Given load factor of N/M.
  - Average runtime is Θ(L).
  - Average isn’t the whole story. Want balanced buckets. Analogous to maintaining bushiness in a BST, but conceptually much easier to solve.



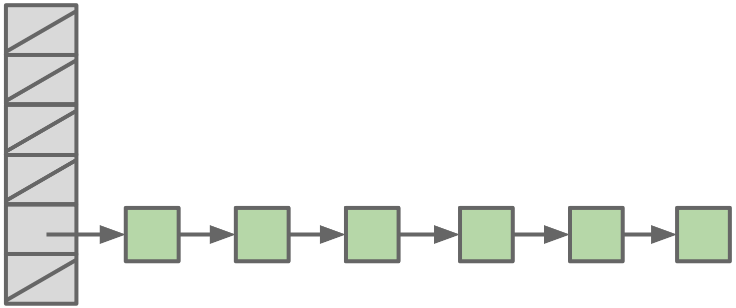


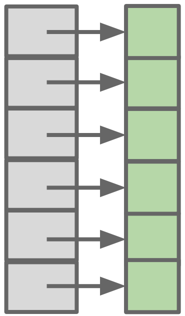
Hash Functions

What Makes a good .hashCode()?

Goal: We want hash tables that look like the table on the right.

- Want a hashCode that spreads things out nicely on real data.
  - Example #1: return 0 is a bad hashCode function.
  - Example #2: Our convertToInt function for strings was bad. Top bits were ignored, e.g. “potato” and “give me a potato” have same hashCode.
- Writing a good hashCode() method can be tricky.





Example: String hashCode function

Our convertToInt function:

- $h(s) = (s_0 - \text{'a'} + 1) \times 32^{n-1} + (s_1 - \text{'a'} + 1) \times 32^{n-2} + \ldots + (s_{n-1} - \text{'a'} + 1)$

Problems:

- Intended for lower case strings only.
- Top bits are totally ignored.

Example: String hashCode function

Improved convertToInt function:

- $h(s) = s_0 \times 32^{n-1} + s_1 \times 32^{n-2} + \ldots + s_{n-1}$

Problems:

- Intended for lower case strings only: Fix by removing - ‘a’
- Top bits are totally ignored.
  - Why? Because multiplying by 32 is equivalent to left shifting by 5 bits. Result: Top characters get pushed out completely.
  - How can we fix?

Example: String hashCode function

Java’s actual hashCode() function for Strings:

- $h(s) = s_0 \times 31^{n-1} + s_1 \times 31^{n-2} + \ldots + s_{n-1}$

Problems:

- Intended for lower case strings only: Fix by removing - ‘a’
- Top bits are totally ignored.
  - Why? Because multiplying by 32 is equivalent to left shifting by 5 bits. Result: Top characters get pushed out completely.
  - How can we fix: Multiply by powers of 31!
  - In convertToInt, we tried to have the kth character contribute to specific bits of the hashCode(). Nice for understanding lecture, but exactly the wrong idea for avoiding hashCode collisions!

Example: String hashCode Function Example

Java's hashCode() function:

- $h(s) = s_0 \times 31^{n-1} + s_1 \times 31^{n-2} + \ldots + s_{n-1}$

Food for thought: Hash tables with a number of buckets equal to a multiple of 31 will not work very well with this hashCode(). Why?

Consider 'cat':

convertToInt()

$$\texttt{00 00000 00000 00000 } \underbrace{\texttt{00011}}_{c}\ \underbrace{\texttt{00001}}_{a}\ \underbrace{\texttt{10100}}_{t}$$

'c'	00000000 01100011	*31^2	95139	00 00000 00000 00010 11100 11101 00011
'a'	00000000 01100001	*31	3007	00 00000 00000 00000 00010 11101 11111
't'	00000000 01110100	*1	116	00 00000 00000 00000 00000 00011 10100
			98262	00 00000 00000 00010 11111 11110 10110

Why these numbers? [ASCII]

Hashing

How do you make hashbrowns?

- Chopping potato into nice predictable segments? No way!
- This is a hashbrown:

Example: Hashing a Collection

Lists are a lot like strings: Collection of items each with its own hashCode:

```
@Override
public int hashCode() {
   int hashCode = 1;
   for (Object o : this) {
       hashCode = hashCode * 31;
       hashCode = hashCode + o.hashCode();
    }
    return hashCode;
}
```

elevate/smear the current hash code

add new item's hash code

To save time hashing: Look at only first few items.

- Higher chance of collisions but things will still work.

Example: Hashing a Recursive Data Structure

Computation of the hashCode of a recursive data structure involves recursive computation.

- For example, binary tree hashCode (assuming sentinel leaves):

```
@Override
public int hashCode() {
    if (this.value == null) {
        return 0;
    }
    return  this.value.hashCode() +
     31 * this.left.hashCode() +
     31 * 31 * this.right.hashCode();
}
```

Default hashCodes()

All Objects have hashCode() function.

- Default: returns `this` (i.e. address of object).
  - Can have strange consequences: “hello”.hashCode() is not the same as (“h” + “ello”).hashCode()
- Can override for your type.
- Hash tables (HashSet, HashMap, etc.) are so important that Java requires that all objects implement hashCode().

HashSets and HashMaps

Java provides a hash table based implementation of sets and maps.

- Idea is very similar to what we've done in lecture.
- Warning: Never store mutable objects in a `HashSet` or `HashMap`!
- Warning #2: Never override `equals` without also overriding `hashCode`.
  - Why these warnings? See study guide.

In lab 9, you'll get a chance to implement a hash map.

Summary

With good hashCode() and resizing, operations are $\Theta(1)$ amortized.

- No need to maintain bushiness (but still need good hashCode).
- Store and retrieval does not require items to be comparable.

	contains(x)	insert(x)
Linked List	$\Theta(N)$	$\Theta(N)$
Bushy BSTs	$\Theta(\log N)$	$\Theta(\log N)$
Unordered Array	$\Theta(N)$	$\Theta(N)$
Hash Table	$\Theta(1)$	$\Theta(1)$

Worst case runtimes

Used by: TreeSet

Used by: HashSet, Python dictionaries

Collision Resolution (Extra)

Open Addressing: An Alternate Disambiguation Strategy (Extra)

If target bucket is already occupied, use a different bucket, e.g.

- Linear probing: Use next address, and if already occupied, just keep scanning one by one.
    - Demo: http://goo.gl/o5EDvb
- Quadratic probing: Use next address, and if already occupied, try looking 4 ahead, then 9 ahead, then 16 ahead, ...
- Many more possibilities. See the optional reading for today (or CS170) for a more detailed look.

In 61B, we'll settle for external chaining.

Citations

http://www.nydailynews.com/news/national/couple-calls-911-forgotten-mcdonalds-hash-browns-article-1.1543096

http://en.wikipedia.org/wiki/Pigeonhole_principle#mediaviewer/File:TooManyPigeons.jpg

https://cookingplanit.com/public/uploads/inventory/hashbrown_1366322674.jpg