CS61B, 2019

Lecture 19: Hashing

- Set Implementations, DataIndexedIntegerSet
- Integer Representations of Strings, Integer Overflow
- Hash Tables and Handling Collisions
- Hash Table Performance and Resizing
- Hash Tables in Java

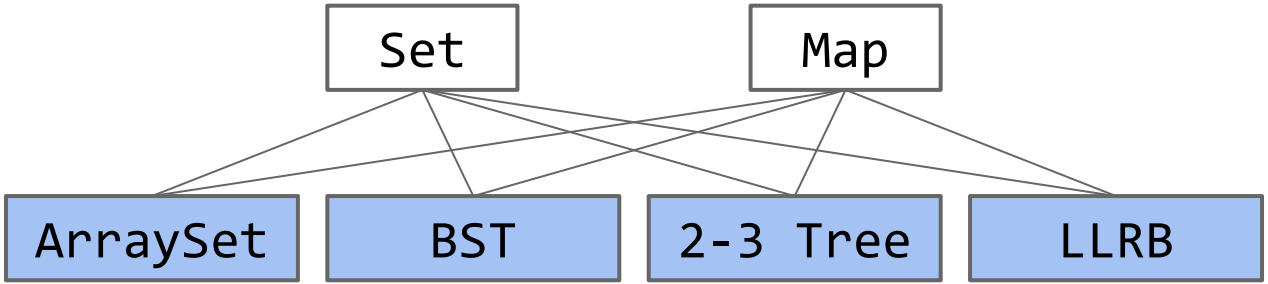
datastructur.es

Data Indexed Arrays

datastructur.es

Sets

We've now seen several implementations of the Set (or Map) ADT.



Worst case runtimes

	contains(x)	add(x)	Notes
ArraySet	Θ(N)	Θ(N)	
BST	Θ(N)	Θ(N)	Random trees are Θ(log N).
2-3 Tree	Θ(log N)	Θ(log N)	Beautiful idea. Very hard to implement.
LLRB	Θ(log N)	Θ(log N)	Maintains bijection with 2-3 tree. Hard to implement.

datastructur.es

Limits of Search Tree Based Sets

Our search tree based sets require items to be comparable.

- Need to be able to ask “is X < Y?” Not true of all types.
- Could we somehow avoid the need for objects to be comparable?

Our search tree sets have excellent performance, but could maybe be better?

- Θ(log N) is amazing. 1 billion items is still only height ~30.
- Could we somehow do better than Θ(log N)?

Today we’ll see the answer to both of the questions above is yes.

datastructur.es

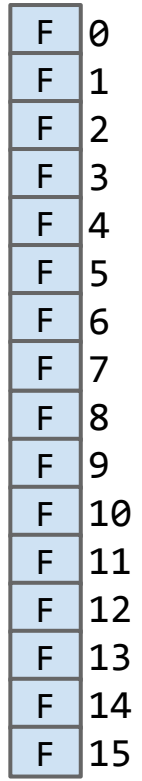
Using Data as an Index

One extreme approach: Create an array of booleans indexed by data!

- Initially all values are false.
- When an item is added, set appropriate index to true.

```
DataIndexedIntegerSet diis;
diis = new DataIndexedIntegerSet();
diis.add(0);
```

Set containing nothing



datastructur.es

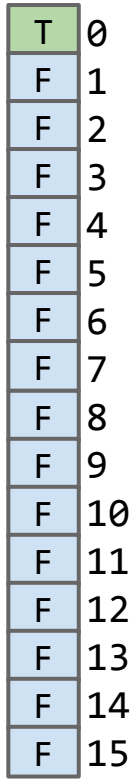
Using Data as an Index

One extreme approach: Create an array of booleans indexed by data!

- Initially all values are false.
- When an item is added, set appropriate index to true.

```
DataIndexedIntegerSet diis;
diis = new DataIndexedIntegerSet();
diis.add(0);
```

Set containing 0

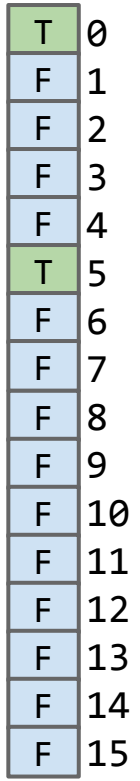


datastructur.es

Using Data as an Index

One extreme approach: Create an array of booleans indexed by data!

- Initially all values are false.
- When an item is added, set appropriate index to true.




```
DataIndexedIntegerSet diis;
diis = new DataIndexedIntegerSet();
diis.add(0);
diis.add(5);
```

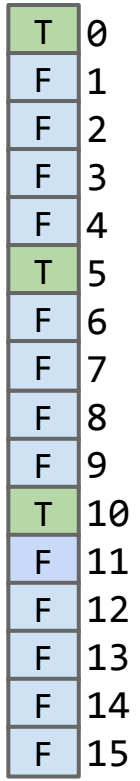
Set containing 0, 5

datastructur.es

Using Data as an Index

One extreme approach: Create an array of booleans indexed by data!

- Initially all values are false.
- When an item is added, set appropriate index to true.




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

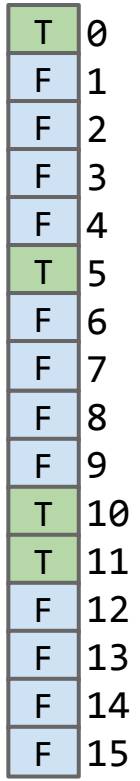
Set containing 0, 5, 10

datastructur.es

Using Data as an Index

One extreme approach: Create an array of booleans indexed by data!

- Initially all values are false.
- When an item is added, set appropriate index to true.




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

Set containing 0, 5, 10, 11

datastructur.es

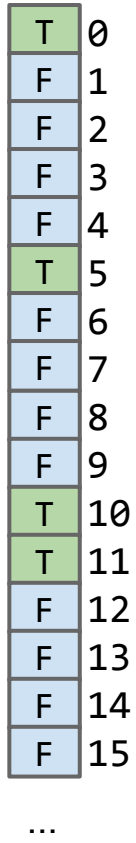
DataIndexedIntegerSet Implementation

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

    public DataIndexedIntegerSet() {
        present = new boolean[2000000000];
    }

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

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



Set containing 0, 5, 10, 11

datastructur.es

DataIndexedIntegerSet Implementation

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

    public DataIndexedIntegerSet() {
        present = new boolean[2000000000];
    }

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

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

	contains(x)	add(x)
ArraySet	Θ(N)	Θ(N)
BST	Θ(N)	Θ(N)
2-3 Tree	Θ(log N)	Θ(log N)
LLRB	Θ(log N)	Θ(log N)
DataIndexedArray	Θ(1)	Θ(1)

datastructur.es

Using Data as an Index

One extreme approach: Create an array of booleans indexed by data!

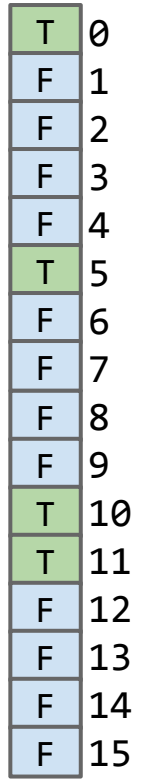
- Initially all values are false.
- When an item is added, set appropriate index to true.

Downsides of this approach (that we will try to address):

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

Set containing 0, 5, 10, 11



datastructur.es

DataIndexedEnglishWordSet

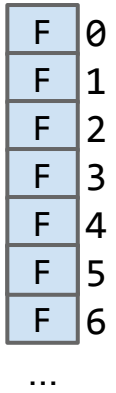
datastructur.es

Generalizing the DataIndexedIntegerSet Idea

Ideally, we want a data indexed set that can store arbitrary types.

```
DataIndexedSet<String> dis =
    new DataIndexedSet<>();
dis.add("cat");
dis.add("bee");
dis.add("dog");
```

Where do cat, bee, and dog go???



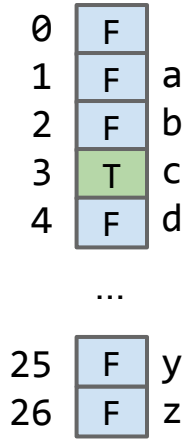
But previous idea only supports integers!

- Let's talk about storing Strings.
- Will get to generic types later.

datastructur.es
CC BY NC SA

Generalizing the DataIndexedIntegerSet Idea

Suppose we want to add("cat")



The key question:

- What is the catth element of a list?
- One idea: Use the first letter of the word as an index.
  - a = 1, b = 2, c = 3, …, z = 26

What's wrong with this approach?

- Other words start with c.
  - contains("chupacabra") : true
- Can't store "=98yae98fwyawef"

"chupacabra" **collides with** "cat"

datastructur.es

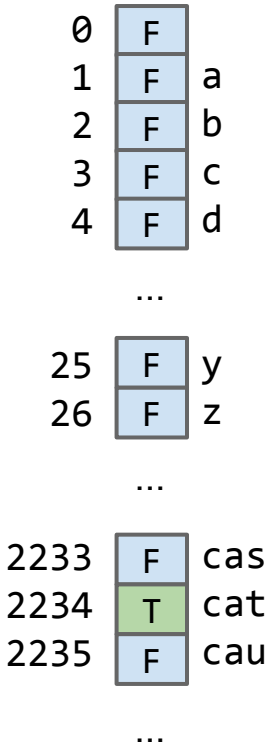
Avoiding Collisions

Use all digits by multiplying each by a power of 27.

- a = 1, b = 2, c = 3, …, z = 26
- Thus the index of "cat" is $(3 \times 27^{2}) + (1 \times 27^{1}) + (20 \times 27^{0})$ = 2234.

Why this specific pattern?

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



datastructur.es

The Decimal Number System vs. Our System for Strings

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

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

Our system for strings is almost the same, but with letters.

datastructur.es

Test Your Understanding

Convert the word “bee” into a number by using our “powers of 27” strategy.

Reminder: $cat_{27} = (3 \times 27^{2}) + (1 \times 27^{1}) + (20 \times 27^{0}) = 2234_{10}$

Hint: ‘b’ is letter 2, and ‘e’ is letter 5.

datastructur.es

Test Your Understanding

Convert the word “bee” into a number by using our “powers of 27” strategy.

Reminder: $cat_{27} = (3 \times 27^{2}) + (1 \times 27^{1}) + (20 \times 27^{0}) = 2234_{10}$

Hint: ‘b’ is letter 2, and ‘e’ is letter 5.

- $bee_{27} = (2 \times 27^{2}) + (5 \times 27^{1}) + (5 \times 27^{0}) = 1598_{10}$

datastructur.es

Uniqueness

- $cat_{27} = (3 \times 27^{2}) + (1 \times 27^{1}) + (20 \times 27^{0}) = 2234_{10}$
- $bee_{27} = (2 \times 27^{2}) + (5 \times 27^{1}) + (5 \times 27^{0}) = 1598_{10}$

As long as we pick a base ≥ 26, this algorithm is guaranteed to give each lowercase English word a unique number!

- Using base 27, no other words will get the number 1598.

In other words: Guaranteed that we will never have a collision.

datastructur.es

Implementing `englishToInt` (optional)

Optional exercise: Try to write a function `englishToInt` that can convert English strings to integers by adding characters scaled by powers of 27.

Examples:

- a: 1
- z: 26
- aa: 28
- bee: 1598
- cat: 2234
- dog: 3328
- potato: 237,949,071

datastructur.es

Implementing englishToInt (optional) (solution)

```
/** 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 englishToInt(String s) {
    int intRep = 0;
    for (int i = 0; i < s.length(); i += 1) {
        intRep = intRep * 27;
        intRep = intRep + letterNum(s, i);
    }
    return intRep;
}
```

datastructur.es

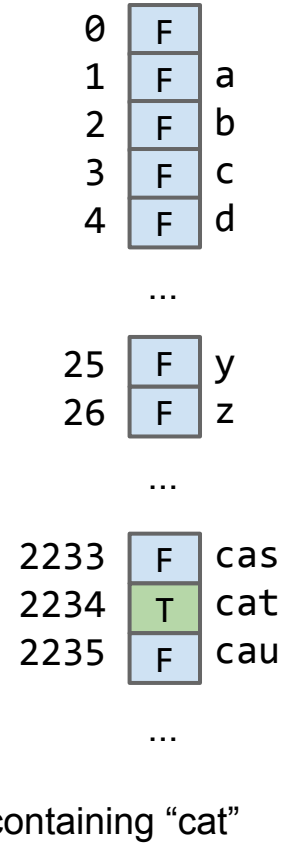
DataIndexedEnglishWordSet Implementation

```
public class DataIndexedEnglishWordSet {
    private boolean[] present;

    public DataIndexedEnglishWordSet() {
        present = new boolean[2000000000];
    }

    public void add(String s) {
        present[englishToInt(s)] = true;
    }

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



Set containing “cat”

datastructur.es

DataIndexedStringSet

datastructur.es

DataIndexedStringSet

Using only lowercase English characters is too restrictive.

- What if we want to store strings like “2pac” or “eGg!”?
- To understand what value we need to use for our base, let’s discuss briefly discuss the ASCII standard.

CC BY NC SA

ASCII Characters

The most basic character set used by most computers is ASCII format.

- Each possible character is assigned a value between 0 and 127.
- Characters 33 - 126 are “printable”, and are shown below.
- For example, `char c = 'D'` is equivalent to `char c = 68`.

33	!
34	"
35	#
36	$
37	%
38	&
39	'
40	(
41	)
42	*
43	+
44	,
45	-
46	.
47	/
48	0

49	1
50	2
51	3
52	4
53	5
54	6
55	7
56	8
57	9
58	:
59	;
60	<
61	=
62	>
63	?
64	@

65	A
66	B
67	C
68	D
69	E
70	F
71	G
72	H
73	I
74	J
75	K
76	L
77	M
78	N
79	O
80	P

81	Q
82	R
83	S
84	T
85	U
86	V
87	W
88	X
89	Y
90	Z
91	[
92	\
93	]
94	^
95	_
96	`

97	a
98	b
99	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o
112	p

113	q
114	r
115	s
116	t
117	u
118	v
119	w
120	x
121	y
122	z
123	{
124	|
125	}
126	~

biggest value is 126

datastructur.es

DataIndexedStringSet

Using only lowercase English characters is too restrictive.

- What if we want to store strings like “2pac” or “eGg!”?
- Maximum possible value for english-only text including punctuation is 126, so let’s use 126 as our base in order to ensure unique values for all possible strings.

Examples:

- $bee_{126} = (98 \times 126^{2}) + (101 \times 126^{1}) + (101 \times 126^{0}) = 1{,}568{,}675$
- $2pac_{126} = (50 \times 126^{3}) + (112 \times 126^{2}) + (97 \times 126^{1}) + (99 \times 126^{0}) = 101{,}809{,}233$
- $eGg!_{126} = (98 \times 126^{3}) + (71 \times 126^{2}) + (98 \times 126^{1}) + (33 \times 126^{0}) = 203{,}178{,}213$

datastructur.es

Implementing `asciiToInt`

The corresponding integer conversion function is actually even simpler than `englishToInt`! Using the raw character value means we avoid the need for a helper method.

```
public static int asciiToInt(String s) {
    int intRep = 0;
    for (int i = 0; i < s.length(); i += 1) {
        intRep = intRep * 126;
        intRep = intRep + s.charAt(i);
    }
    return intRep;
}
```

What if we want to use characters beyond ASCII?

datastructur.es

Going Beyond ASCII

chars in Java also support character sets for other languages and symbols.

- `char c = '☔'` is equivalent to `char c = 9730`.
- `char c = '鳌'` is equivalent to `char c = 40140`.
- `char c = '혜'` is equivalent to `char c = 54812`.
- This encoding is known as Unicode. Table is too big to list.

datastructur.es

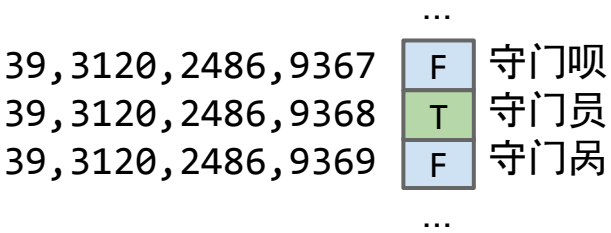
Example: Computing Unique Representations of Chinese

The largest possible value for Chinese characters is 40,959*, so we'd need to use this as our base if we want to have a unique representation for all possible strings of Chinese characters.

Example:

- $守门员_{40959} = (23432 \times 40959^{2}) + (38376 \times 40959^{1}) + (21592 \times 40959^{0}) = 39,312,024,869,368$

*If you're curious, the last character is: 鼐



datastructur.es

Integer Overflow and Hash Codes

datastructur.es

Major Problem: Integer Overflow

In Java, the largest possible integer is 2,147,483,647.

- If you go over this limit, you overflow, starting back over at the smallest integer, which is -2,147,483,648.
- In other words, the next number after 2,147,483,647 is -2,147,483,648.

```
int x = 2147483647;
System.out.println(x);
System.out.println(x + 1);
```

```
jug ~/Dropbox/61b/lec/hashing
$ javac BiggestPlusOne.java
$ java BiggestPlusOne
2147483647
-2147483648
```

datastructur.es

Consequence of Overflow: Collisions

Because Java has a maximum integer, we won't get the numbers we expect!

- With base 126, we will run into overflow even for short strings.
  - Example: $\text{omens}_{126} = 28{,}196{,}917{,}171$, which is much greater than the maximum integer!
  - `asciiToInt('omens')` will give us -1,867,853,901 instead.

datastructur.es

Consequence of Overflow: Collisions

Because Java has a maximum integer, we won't get the numbers we expect!

- With base 126, we will run into overflow even for short strings.
  - Example: omens_{126} = 28,196,917,171, which is much greater than the maximum integer!
  - `asciiToInt('omens')` will give us -1,867,853,901 instead.

Overflow can result in **collisions**, causing incorrect answers due.

```
public void moo() {
  DataIndexedStringSet disi = new DataIndexedStringSet();
  disi.add("melt banana");
  disi.contains("subterrestrial anticosmetic");
  //asciiToInt for these strings is 839099497
}
```

returns true!

datastructur.es

Hash Codes and the Pigeonhole Principle

The official term for the number we're computing is "hash code".

- Via Wolfram Alpha: a hash code "projects a value from a set with many (or even an infinite number of) members to a value from a set with a fixed number of (fewer) members."
- Here, our target set is the set of Java integers, which is of size 4294967296.

datastructur.es

Hash Codes and the Pigeonhole Principle

The official term for the number we're computing is "hash code".

- Via Wolfram Alpha: a hash code "projects a value from a set with many (or even an infinite number of) members to a value from a set with a fixed number of (fewer) members."
- Here, our target set is the set of Java integers, which is of size 4294967296.

Pigeonhole principle tells us that if there are more than 4294967296 possible items, multiple items will share the same hash code.

- There are more than 4294967296 planets.
  - Each has mass, xPos, yPos, xVel, yVel, imgName.
- There are more than 4294967296 strings.
  - "one", "two", … "nineteen quadrillion", …

Bottom line: Collisions are inevitable.

datastructur.es

Two Fundamental Challenges

Two fundamental challenges:

- How do we resolve hashCode collisions (“melt banana” vs. “subterranean anticosmetic”)?
  - We’ll call this ***collision handling***.

- How do we compute a hash code for arbitrary objects?
  - We’ll call this ***computing a hashCode***.
  - Example: Our hashCode for “melt banana” was 839099497.
  - For Strings, this was relatively straightforward (treat as a base 27 or base 126 or base 40959 number).

datastructur.es

Hash Tables: Handling Collisions

datastructur.es

Resolving Ambiguity

Pigeonhole principle tells us that collisions are inevitable due to integer overflow.

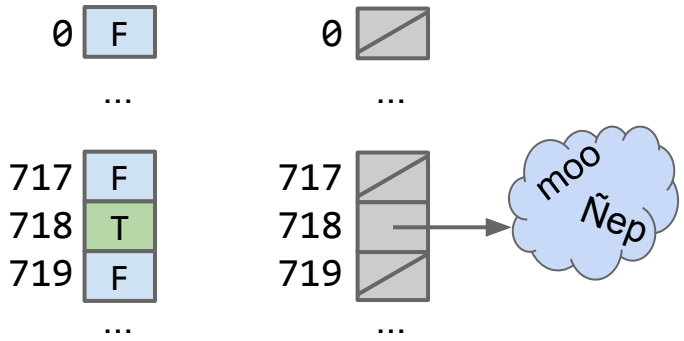
- Example: hash code for “moo” and “Ñep” might both be 718.

Suppose N items have the same numerical representation h:

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

How to implement a “bucket”?

- Conceptually simplest way: LinkedList.
- Could also use ArrayLists.
- Could also use an ArraySet.
- Will see it doesn’t really matter what you do.



datastructur.es

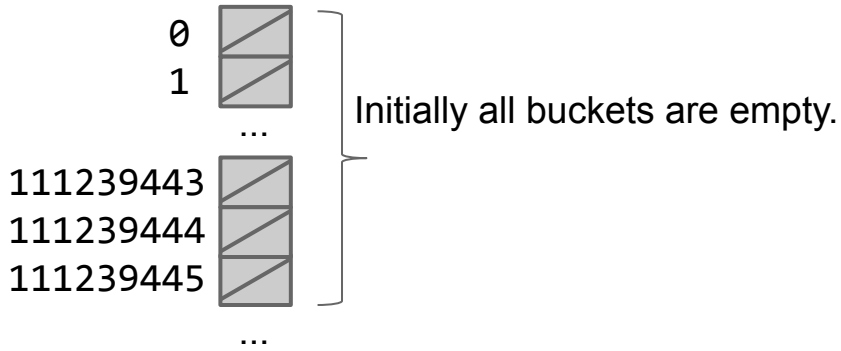
The Separate Chaining Data Indexed Array

Each bucket in our array is initially empty. When an item x gets added at index h:

- If bucket h is empty, we create a new list containing x and store it at index h.
- If bucket h is already a list, we add x to this list if it is not already present.

We might call this a “separate chaining data indexed array”.

- Bucket #h is a “separate chain” of all items that have hash code h.



datastructur.es

The Separate Chaining Data Indexed Array

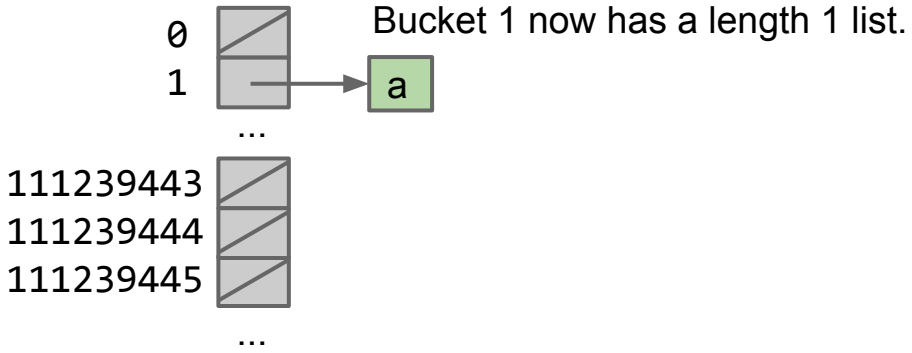
Each bucket in our array is initially empty. When an item x gets added at index h:

- If bucket h is empty, we create a new list containing x and store it at index h.
- If bucket h is already a list, we add x to this list if it is not already present.

We might call this a “separate chaining data indexed array”.

- Bucket #h is a “separate chain” of all items that have hash code h.

add(“a”)



datastructur.es

The Separate Chaining Data Indexed Array

Each bucket in our array is initially empty. When an item x gets added at index h:

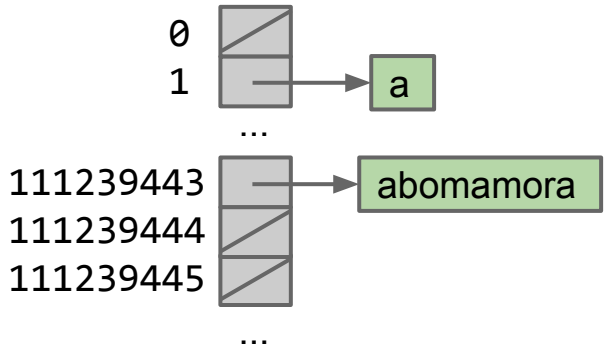
- If bucket h is empty, we create a new list containing x and store it at index h.
- If bucket h is already a list, we add x to this list if it is not already present.

We might call this a “separate chaining data indexed array”.

- Bucket #h is a “separate chain” of all items that have hash code h.

add(“a”)

add(“abomamora”)



datastructur.es

The Separate Chaining Data Indexed Array

Each bucket in our array is initially empty. When an item x gets added at index h:

- If bucket h is empty, we create a new list containing x and store it at index h.
- If bucket h is already a list, we add x to this list if it is not already present.

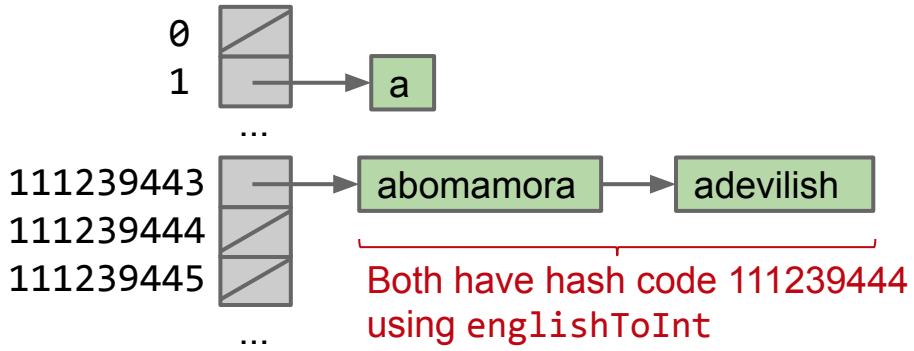
We might call this a “separate chaining data indexed array”.

- Bucket #h is a “separate chain” of all items that have hash code h.

add(“a”)

add(“abomamora”)

add(“adevilish”)



datastructur.es

The Separate Chaining Data Indexed Array

Each bucket in our array is initially empty. When an item x gets added at index h:

- If bucket h is empty, we create a new list containing x and store it at index h.
- If bucket h is already a list, we add x to this list **<u>if it is not already present</u>**.

We might call this a “separate chaining data indexed array”.

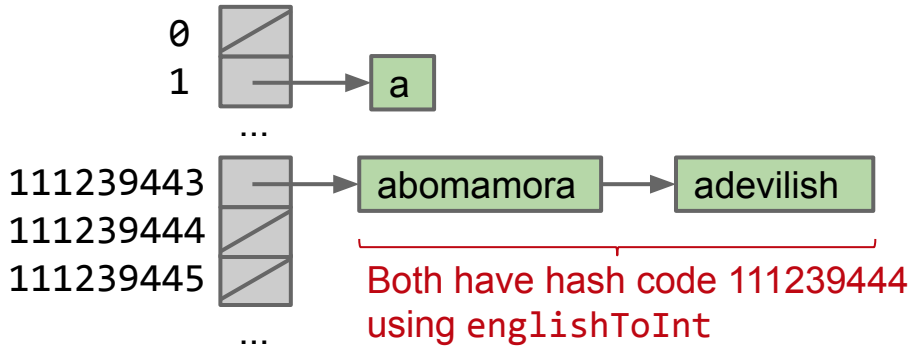
- Bucket #h is a “separate chain” of all items that have hash code h.

add(“a”)

add(“abomamora”)

add(“adevilish”)

add(“abomamora”)



datastructur.es

The Separate Chaining Data Indexed Array

Each bucket in our array is initially empty. When an item x gets added at index h:

- If bucket h is empty, we create a new list containing x and store it at index h.
- If bucket h is already a list, we add x to this list if it is not already present.

We might call this a “separate chaining data indexed array”.

- Bucket #h is a “separate chain” of all items that have hash code h.

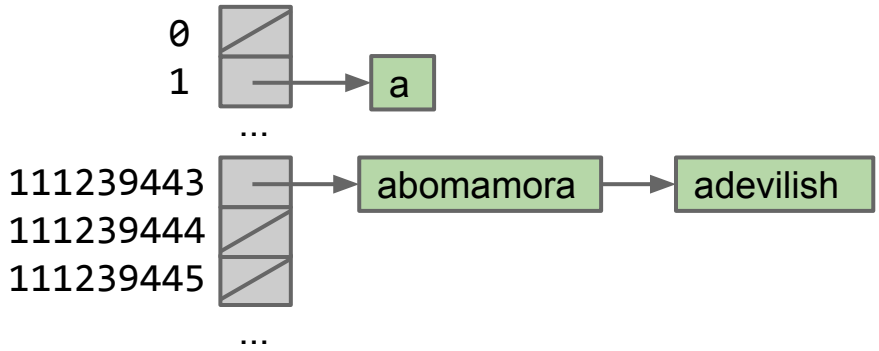
add(“a”)

add(“abomamora”)

add(“adevilish”)

add(“abomamora”)

contains(“adevilish”)

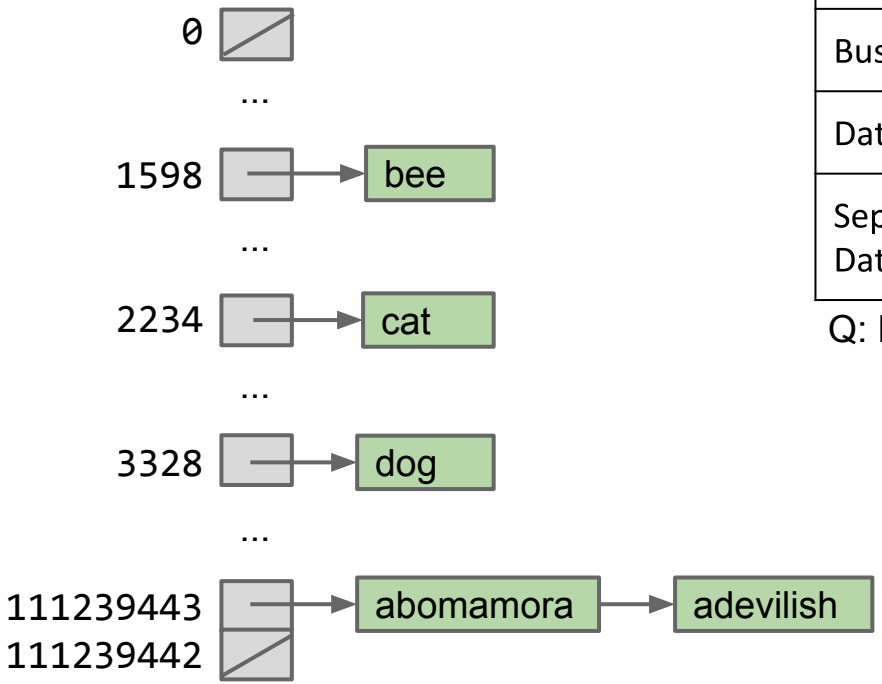


- Look at all items in bucket 111239443 to see if “adevilish” is present.

datastructur.es

Separate Chaining Performance

Observation: Worst case runtime will be proportional to length of longest list.



Worst case time	contains(x)	add(x)
Bushy BSTs	Θ(log N)	Θ(log N)
DataIndexedArray	Θ(1)	Θ(1)
Separate Chaining Data Indexed Array	Θ(Q)	Θ(Q)

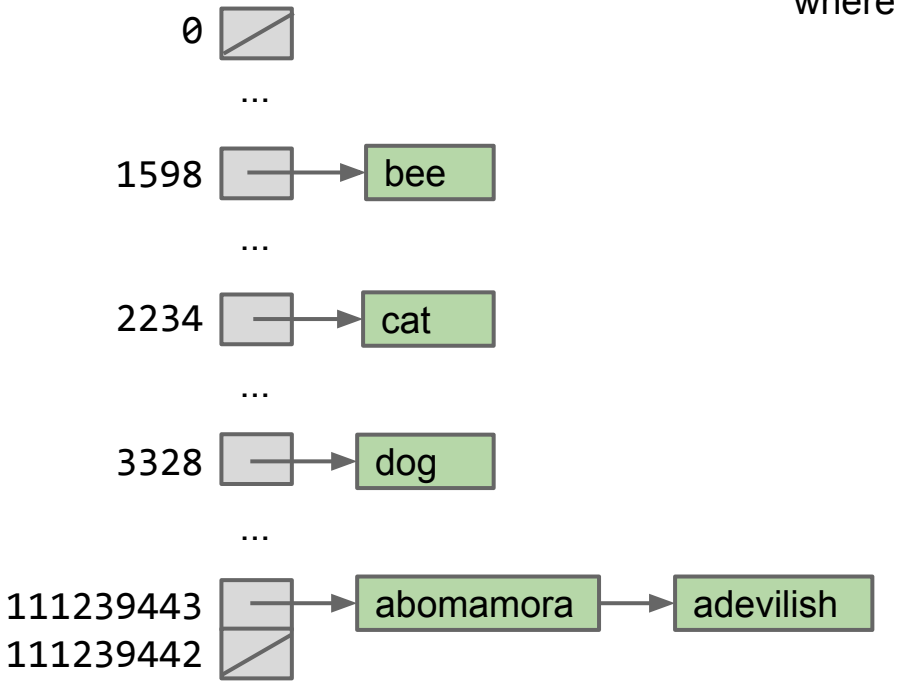
Q: Length of longest list

Why Q and not 1?

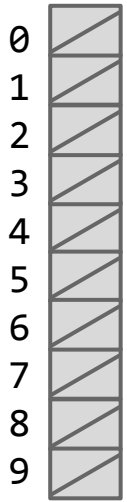
datastructur.es

Saving Memory Using Separate Chaining

Observation: We don't really need billions of buckets.



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

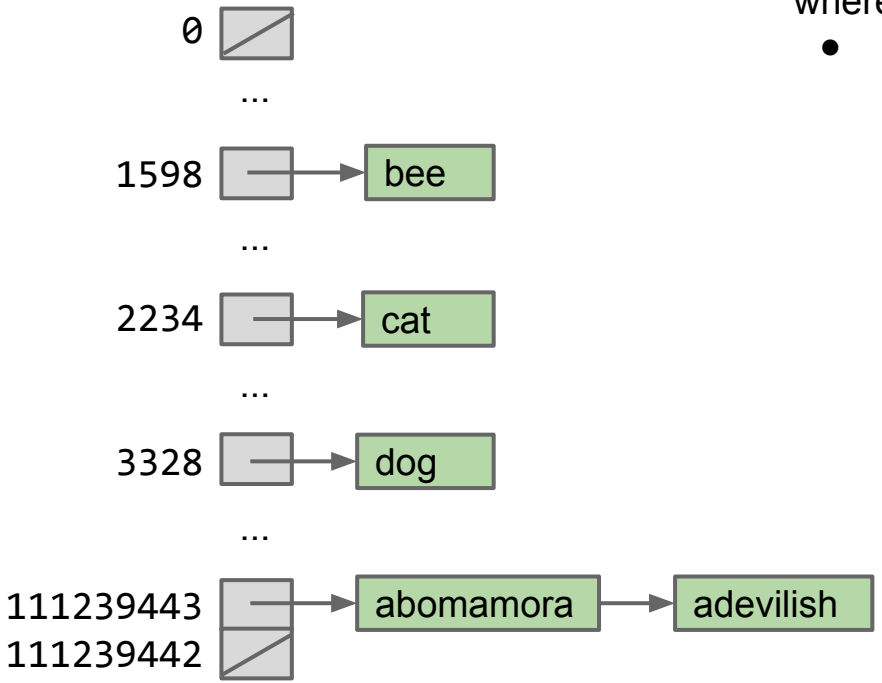


datastructur.es

Saving Memory Using Separate Chaining and Modulus

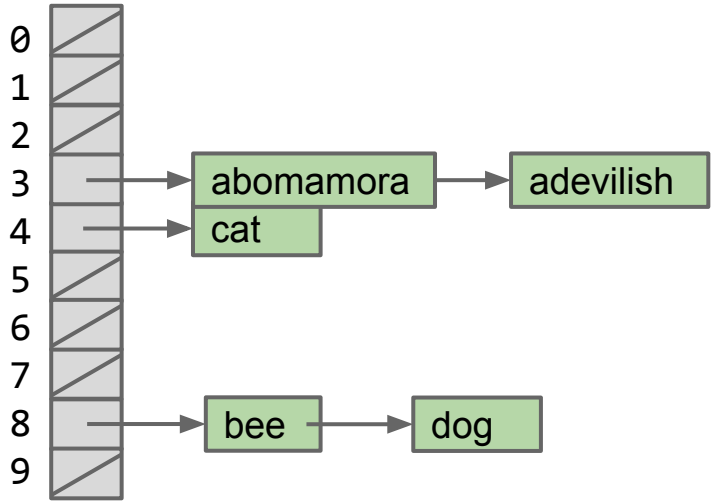
Observation: Can use modulus of hashcode to reduce bucket count.

- Downside: Lists will be longer.



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

- Put in bucket = hashCode % 10

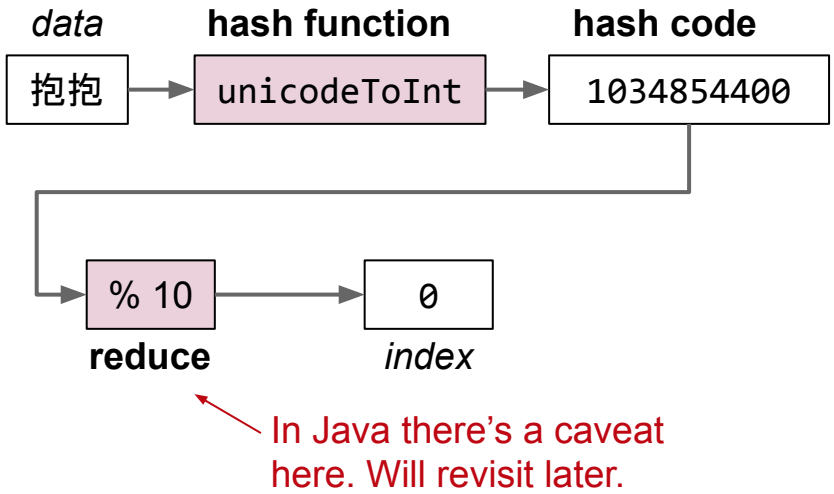


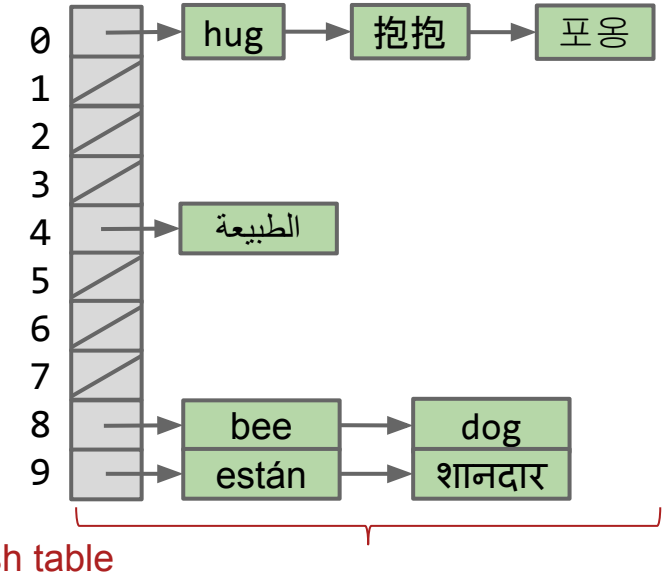
datastructur.es

The Hash Table

What we've just created here is called a **hash table**.

- *Data* is converted by a **hash function** into an integer representation called a **hash code**.
- The **hash code** is then **reduced** to a valid *index*, usually using the modulus operator, e.g. 2348762878 % 10 = 8.



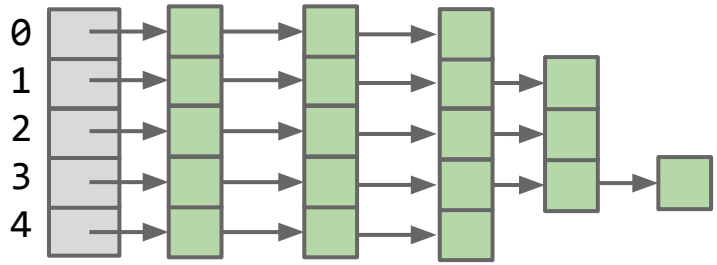


datastructur.es

Hash Table Performance

datastructur.es

Hash Table Runtime



Worst case runtimes

	contains(x)	add(x)
Bushy BSTs	Θ(log N)	Θ(log N)
DataIndexedArray	Θ(1)	Θ(1)
Separate Chaining Hash Table	Θ(Q)	Θ(Q)

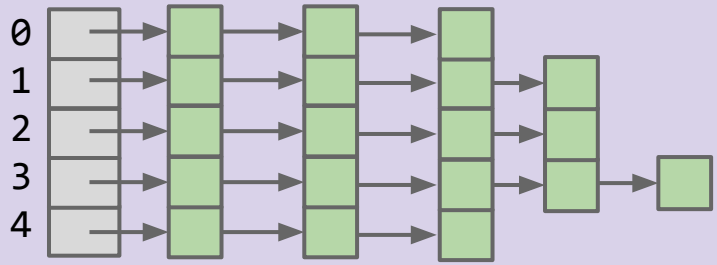
Q: Length of longest list

The good news: We use way less memory and can now handle arbitrary data.

The bad news: Worst case runtime is now Θ(Q), where Q is the length of the longest list.

datastructur.es

Hash Table Runtime: http://yellkey.com/somebody



Worst case runtimes

	contains(x)	add(x)
Bushy BSTs	Θ(log N)	Θ(log N)
DataIndexedArray	Θ(1)	Θ(1)
Separate Chaining Hash Table	Θ(Q)	Θ(Q)

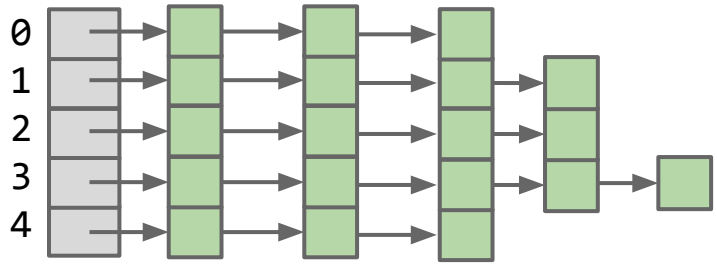
Q: Length of longest list

For the hash table above with 5 buckets, give the order of growth of Q with respect to N.

A. Q is Θ(1)
B. Q is Θ(log N)
C. Q is Θ(N)
D. Q is Θ(N log N)
E. Q is Θ(N^2)

datastructur.es

Hash Table Runtime



Worst case runtimes

	contains(x)	add(x)
Bushy BSTs	Θ(log N)	Θ(log N)
DataIndexedArray	Θ(1)	Θ(1)
Separate Chaining Hash Table	Θ(Q)	Θ(Q)

Q: Length of longest list

For the hash table above with 5 buckets, give the order of growth of Q with respect to N.

C. Q is Θ(N)

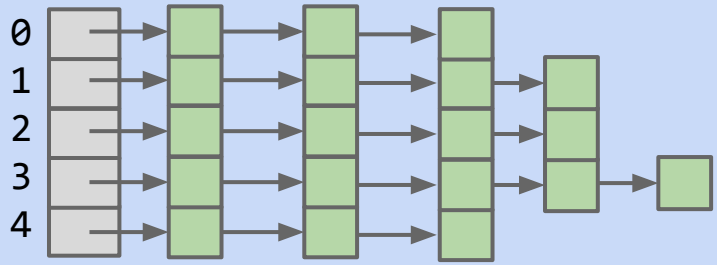
All items evenly distributed.

All items in same list.

In the best case, the length of the longest list will be N/5. In the worst case, it will be N. In both cases, Q(N) is Θ(N).

datastructur.es

Improving the Hash Table



Worst case runtimes

	contains(x)	add(x)
Bushy BSTs	Θ(log N)	Θ(log N)
DataIndexedArray	Θ(1)	Θ(1)
Separate Chaining Hash Table	Θ(Q)	Θ(Q)

Q: Length of longest list

Suppose we have:

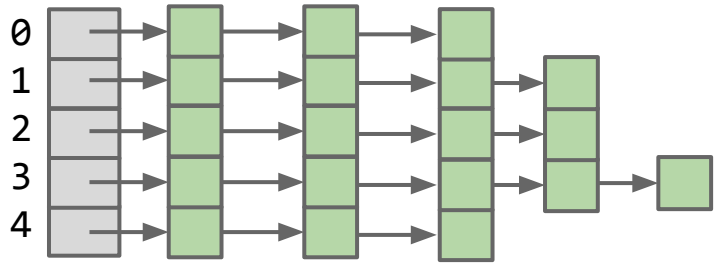
- A fixed number of buckets M.
- An increasing number of items N.

Major problem: Even if items are spread out evenly, lists are of length Q = N/M.

- For M = 5, that means Q = Θ(N). Results in linear time operations.
- Hard question: How can we improve our design to guarantee that N/M is Θ(1)?

datastructur.es

Hash Table Runtime



Worst case runtimes

	contains(x)	add(x)
Bushy BSTs	Θ(log N)	Θ(log N)
DataIndexedArray	Θ(1)	Θ(1)
Separate Chaining Hash Table	Θ(Q)	Θ(Q)

Q: Length of longest list

Suppose we have:

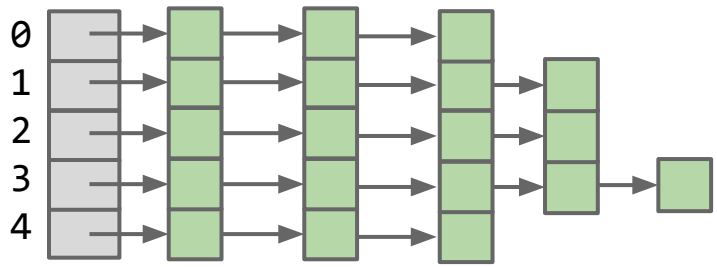
- An increasing number of buckets M.
- An increasing number of items N.

Major problem: Even if items are spread out evenly, lists are of length Q = N/M.

- For M = 5, that means Q = Θ(N). Results in linear time operations.
- Hard question: How can we improve our design to guarantee that N/M is Θ(1)?

datastructur.es

Hash Table Runtime



Suppose we have:

- An increasing number of buckets M.
- An increasing number of items N.

As long as M = Θ(N), then O(N/M) = O(1).

One example strategy: When N/M is ≥ 1.5, then double M.

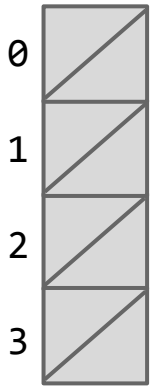
- We often call this process of increasing M “resizing”.
- N/M is often called the “load factor”. It represents how full the hash table is.
- This rule ensures that the average list is never more than 1.5 items long!

datastructur.es

Hash Table Resizing Example

When N/M is ≥ 1.5, then double M.

N = 0 M = 4 N / M = 0

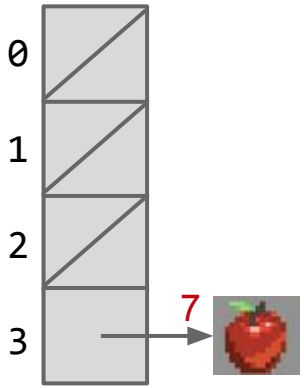


datastructur.es

Hash Table Resizing Example

When N/M is ≥ 1.5, then double M.

N = 1 M = 4 N / M = 0.25

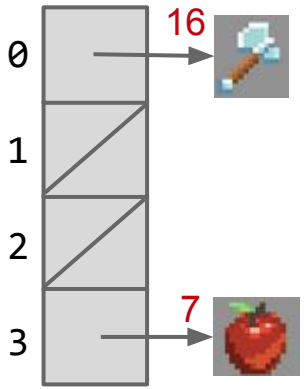


datastructur.es

Hash Table Resizing Example

When N/M is ≥ 1.5, then double M.

N = 2 M = 4 N / M = 0.5

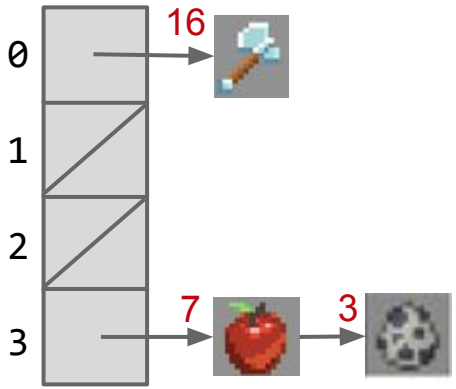


datastructur.es

Hash Table Resizing Example

When N/M is ≥ 1.5, then double M.

N = 3 M = 4 N / M = 0.75

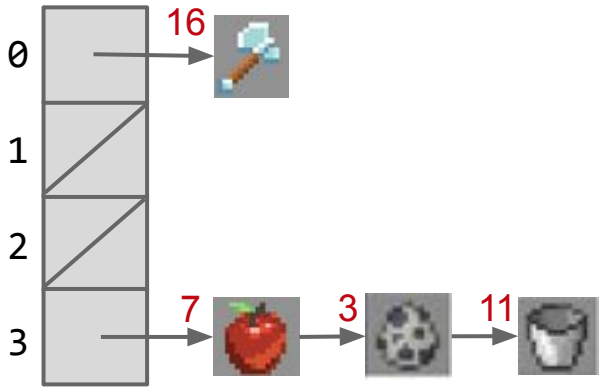


datastructur.es

Hash Table Resizing Example

When N/M is ≥ 1.5, then double M.

N = 4 M = 4 N / M = 1

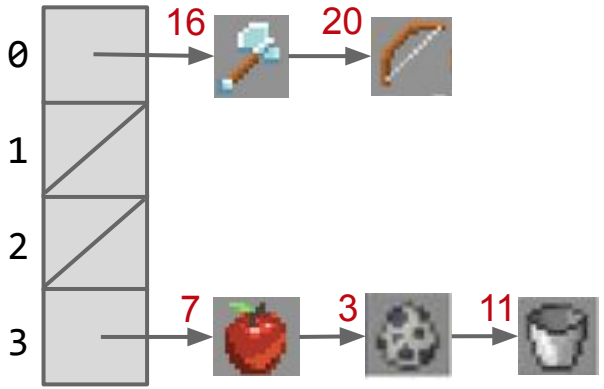


datastructur.es

Hash Table Resizing Example

When N/M is ≥ 1.5, then double M.

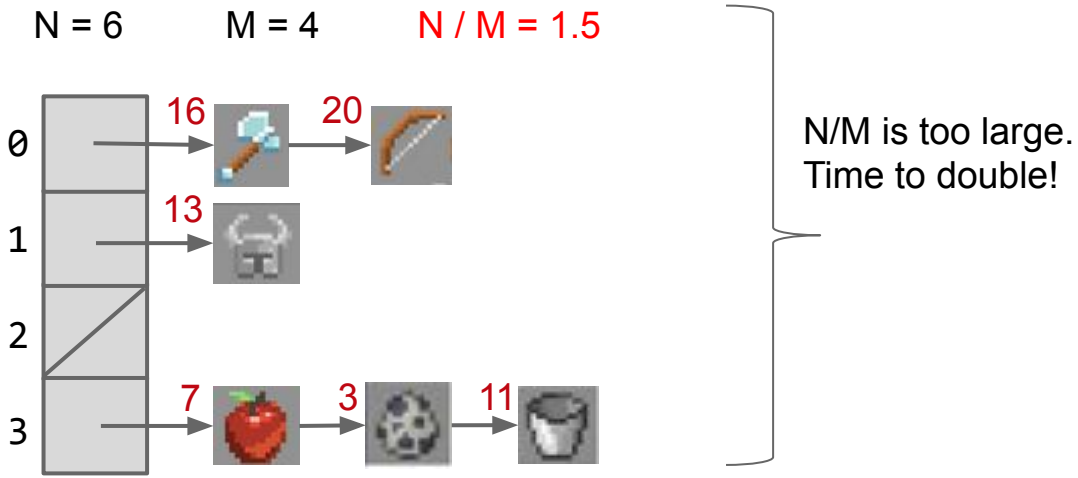
N = 5 M = 4 N / M = 1.25



datastructur.es

Hash Table Resizing Example

When N/M is ≥ 1.5, then double M.

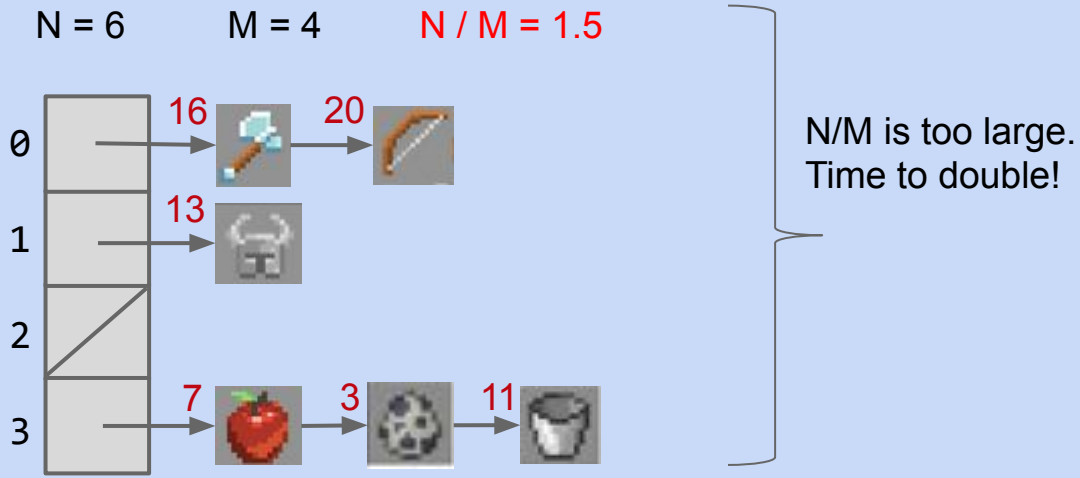


datastructur.es

Hash Table Resizing Example

When N/M is ≥ 1.5, then double M.

- Yellkey question: Where will the bucket go? 
- Or: Draw the results after doubling M.

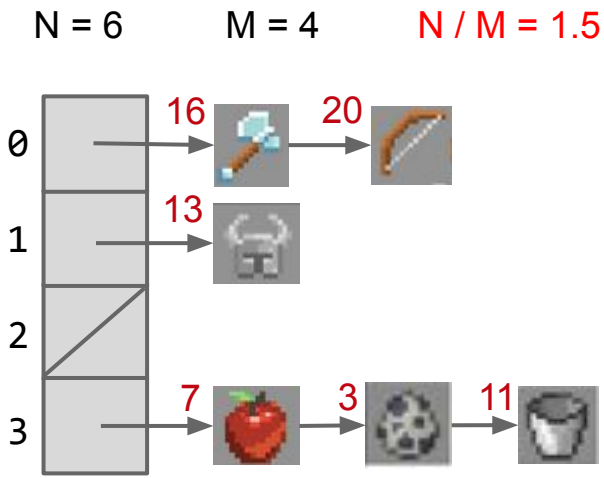


datastructur.es

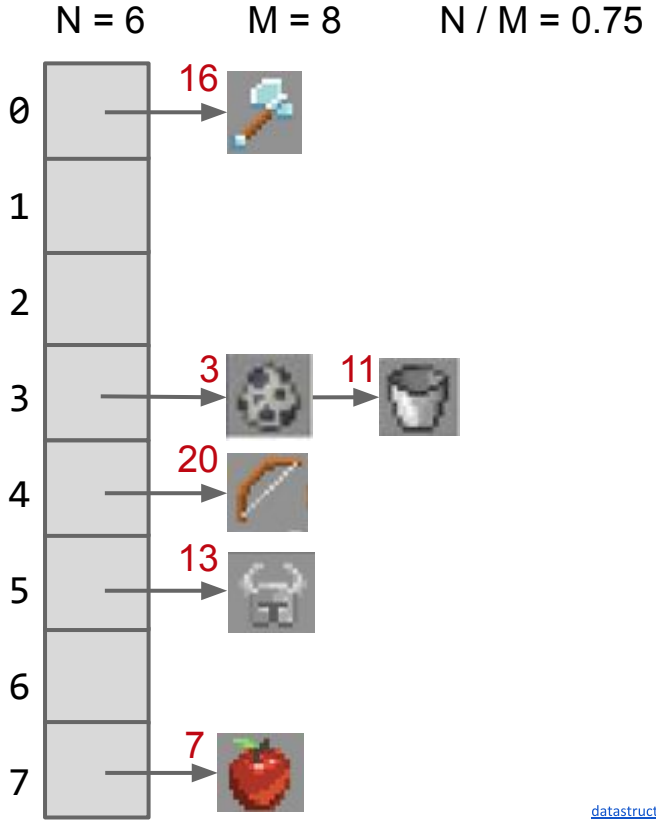
Hash Table Resizing Example

When N/M is ≥ 1.5, then double M.

- Draw the results after doubling M.

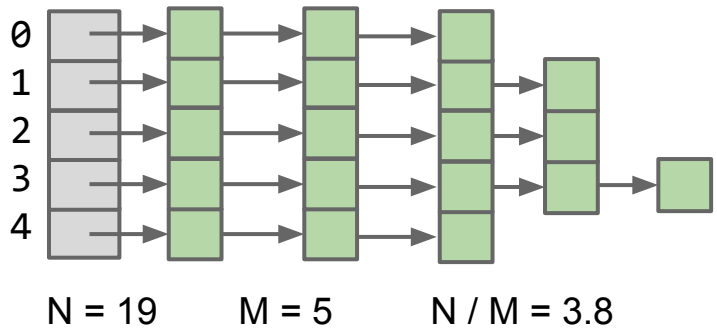


N/M is too large.
Time to double!



datastructur.es

Resizing Hash Table Runtime



Suppose we have:

- An increasing number of buckets M.
- An increasing number of items N.

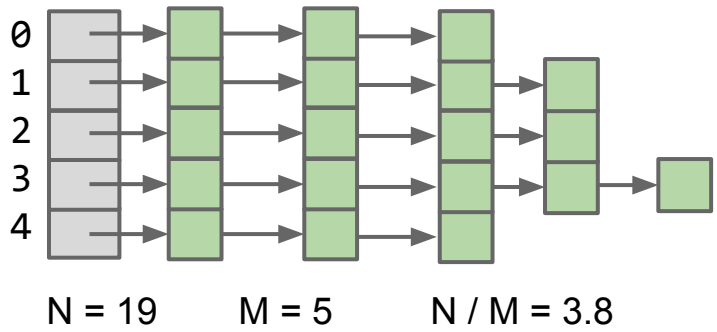
As long as M = Θ(N), then O(N/M) = O(1).

Assuming items are evenly distributed (as above), lists will be approximately N/M items long, resulting in Θ(N/M) runtimes.

- Our doubling strategy ensures that N/M = O(1).
- Thus, worst case runtime for all operations is Θ(N/M) = Θ(1).
  - … unless that operation causes a resize.

datastructur.es

Resizing Hash Table Runtime



Suppose we have:

- An increasing number of buckets M.
- An increasing number of items N.

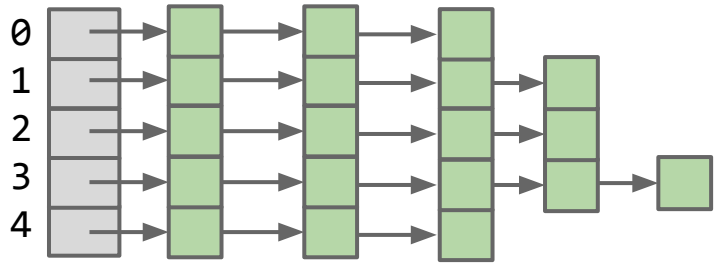
As long as M = Θ(N), then O(N/M) = O(1).

One important thing to consider is the cost of the resize operation.

- Resizing takes Θ(N) time. Have to redistribute all items!
- Most add operations will be Θ(1). Some will be Θ(N) time (to resize).
  - Similar to our ALists, as long as we resize by a multiplicative factor, the average runtime will still be Θ(1).
  - Note: We will eventually analyze this in more detail.

datastructur.es

Hash Table Runtime



Because Q = Θ(N)

Because Q = Θ(1)

Worst case runtimes

	contains(x)	add(x)
Bushy BSTs	Θ(log N)	Θ(log N)
DataIndexedArray	Θ(1)	Θ(1)
Separate Chaining Hash Table With No Resizing	Θ(N)	Θ(N)
… With Resizing	Θ(1)†	Θ(1)*†

*: Indicates “on average”.
†: Assuming items are evenly spread.

Hash table operations are on average constant time if:

- We double M to ensure constant average bucket length.
- Items are evenly distributed.

datastructur.es

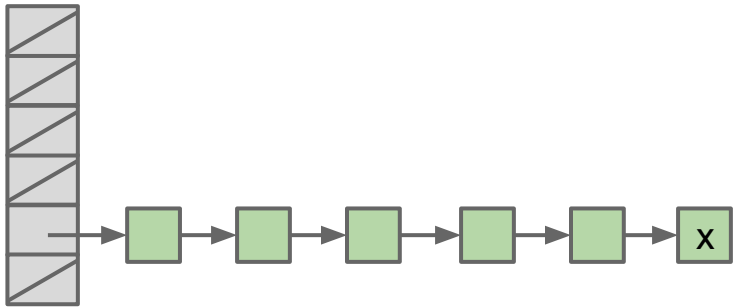
Regarding Even Distribution

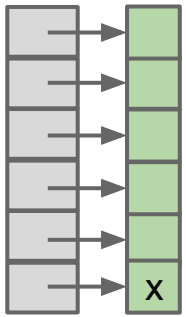
Even distribution of item is critical for good hash table performance.

- Both tables below have load factor of N/M = 1.
- Left table is much worse!
  - Contains is Θ(N) for x.

Will need to discuss how to ensure even distribution.

- First, let's talk a little bit about how hash tables work in Java.





datastructur.es

Hash Tables in Java

datastructur.es

The Ubiquity of Hash Tables

Hash tables are the most popular implementation for sets and maps.

- Great performance in practice.
- Don't require items to be comparable.
- Implementations often relatively simple.
- Python dictionaries are just hash tables in disguise.

In Java, implemented as `java.util.HashMap` and `java.util.HashSet`.

- How does a HashMap know how to compute each object's hash code?
  - Good news: It's not "`implements Hashable`".
  - Instead, all objects in Java must implement a `.hashCode()` method.

datastructur.es

Objects

All classes are hyponyms of `Object`.

- `String toString()`
- **`boolean equals(Object obj)`**
- `Class<?>  getClass()`
- `int hashCode()`
- `protected Object clone()`
- `protected void finalize()`
- `void notify()`
- `void notifyAll()`
- `void wait()`
- `void wait(long timeout)`
- `void wait(long timeout, int nanos)`

datastructur.es

Examples of Real Java HashCodes

We can see that Strings in Java override hashCode, doing something vaguely like what we did earlier.

- Will see the actual hashCode() function later!

```
System.out.println("a".hashCode());
System.out.println("bee".hashCode());
System.out.println("포옹".hashCode());
System.out.println("kamala lifefully".hashCode());
System.out.println("đậu hũ".hashCode());
```

```
jug ~/Dropbox/61b/lec/hashing
$ java JavaHashCodeExamples
"a" 97
"bee" 97410
"포옹" 1732557
"kamala lifefully" 1732557
"đậu hũ" -2108180664
```

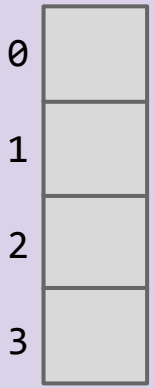
datastructur.es

Using Negative hash codes: http://yellkey.com/without

Suppose that 

's hash code is `-1`.

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

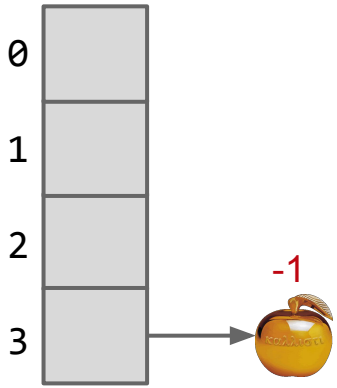


datastructur.es

Using Negative hash codes: http://yellkey.com/medical

Suppose that  's hash code is -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, ...

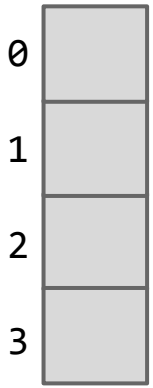


datastructur.es

Using Negative hash codes in Java

Suppose that  ‘s hash code is -1.

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




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

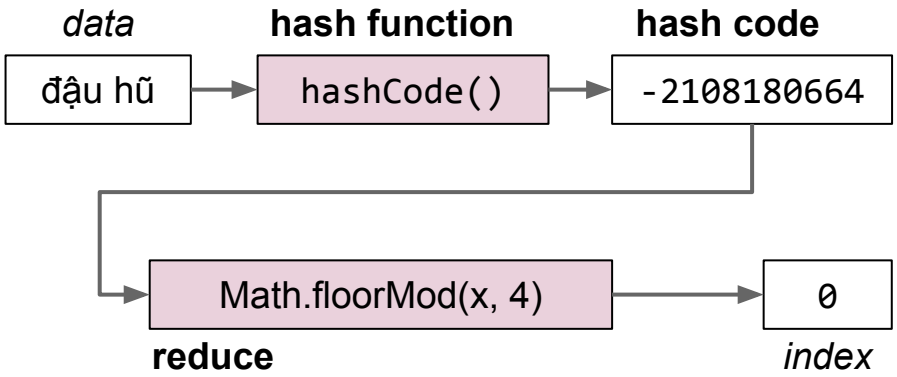
```
$ java ModTest
-1
3
```

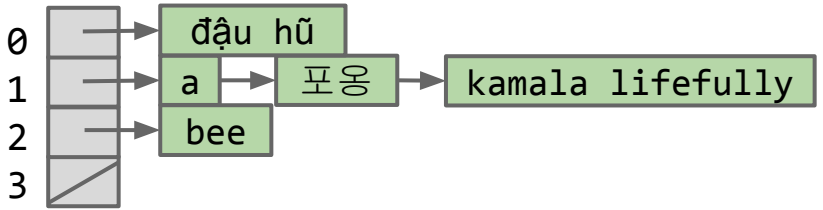
datastructur.es

Hash Tables in Java

Java hash tables:

- *Data* is converted by the **hashCode** method an integer representation called a **hash code**.
- The **hash code** is then **reduced** to a valid *index*, using something like the floorMod function, e.g. Math.floorMod(1732557 % 4) = 8.





datastructur.es

Two Important Warnings When Using HashMaps/HashSets

Warning #1: Never store objects that can change in a `HashSet` or `HashMap`!

- If an object's variables changes, then its hashCode changes. May result in items getting lost.

Warning #2: Never override `equals` without also overriding `hashCode`.

- Can also lead to items getting lost and generally weird behavior.
- HashMaps and HashSets use equals to determine if an item exists in a particular bucket.
- See HW3 or the study guide problems.

datastructur.es

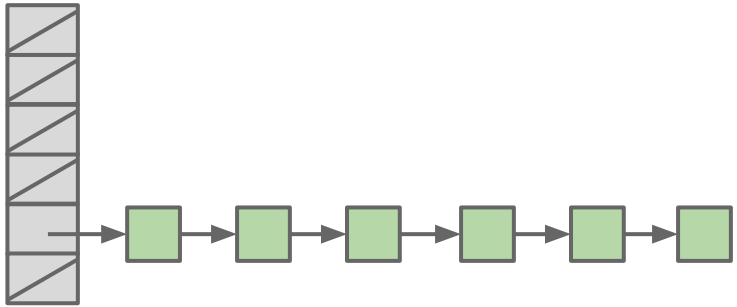
Good HashCodes (Extra)

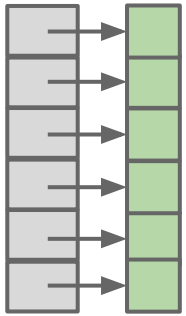
datastructur.es

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: just returning the first character of a word, e.g. “cat” → 3 was also a bad hash function.
  - Example #3: Adding chars together is bad. “ab” collides with “ba”.
  - Example #4: returning string treated as a base B number can be good.
- Writing a good hashCode() method **can be tricky**.





datastructur.es

Hashbrowns and Hash Codes

How do you make hashbrowns?

- Chopping a potato into nice predictable segments? No way!
- Similarly, adding up the characters is not nearly “random” enough.

Can think of multiplying data by powers of some base as ensuring that all the data gets scrambled together into a seemingly random integer.

datastructur.es

Example hashCode Function

The Java 8 hash code for strings. Two major differences from our hash codes:

- Represents strings as a base 31 number.
  - Why such a small base? Real hash codes don't care about uniqueness.
- Stores (caches) calculated hash code so future `hashCode` calls are faster.

```
@Override
public int hashCode() {
    int h = cachedHashValue;
    if (h == 0 && this.length() > 0) {
        for (int i = 0; i < this.length(); i++) {
            h = 31 * h + this.charAt(i);
        }
        cachedHashValue = h;
    }
    return h;
}
```

datastructur.es

Example: Choosing a Base

Java's `hashCode()` function for Strings:

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

Our `asciiToInt` function for Strings:

- $h(s) = s_0 \times 126^{n-1} + s_1 \times 126^{n-2} + \ldots + s_{n-1}$

Which is better?

- Might seem like 126 is better. Ignoring overflow, this ensures a unique numerical representation for all ASCII strings.
- … but overflow is a particularly bad problem for base 126!

datastructur.es

Example: Base 126

Major collision problem:

- “geocronite is the best thing on the earth.”.hashCode() yields 634199182.
- “flan is the best thing on the earth.”.hashCode() yields 634199182.
- “treachery is the best thing on the earth.”.hashCode() yields 634199182.
- “Brazil is the best thing on the earth.”.hashCode() yields 634199182.

Any string that ends in the same last 32 characters has the same hash code.

- Why? Because of overflow.
- Basic issue is that 126^32 = 126^33 = 126^34 = ... 0.
  - Thus upper characters are all multiplied by zero.
  - See CS61C for more.

datastructur.es

Typical Base

A typical hash code base is a small prime.

- Why prime?
  - Never even: Avoids the overflow issue on previous slide.
  - Lower chance of resulting hashCode having a bad relationship with the number of buckets: See study guide problems and hw3.
- Why small?
  - Lower cost to compute.

A full treatment of good hash codes is well beyond the scope of our class.

datastructur.es

Hashbrowns and Hash Codes

How do you make hashbrowns?

- Chopping a potato into nice predictable segments? No way!

Using a prime base yields better “randomness” than using something like base 126.

datastructur.es

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.

datastructur.es

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();
}
```

datastructur.es

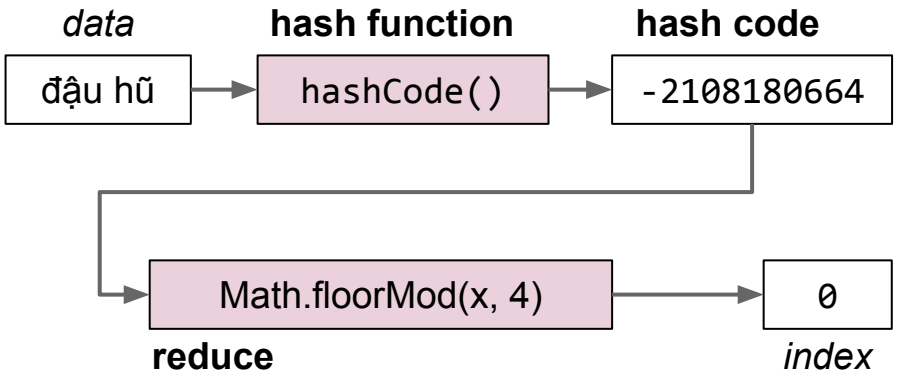
Summary

datastructur.es

Hash Tables in Java

Hash tables:

- *Data* is converted into a hash code.
- The **hash code** is then **reduced** to a valid *index*.
- *Data* is then stored in a bucket corresponding to that *index*.
- Resize when load factor N/M exceeds some constant.
- If items are spread out nicely, you get Θ(1) average runtime.



	contains(x)	add(x)
Bushy BSTs	Θ(log N)	Θ(log N)
Separate Chaining Hash Table With No Resizing	Θ(N)	Θ(N)
… With Resizing	Θ(1)†	Θ(1)*†

*: Indicates “on average”.
†: Assuming items are evenly spread.

datastructur.es

Collision Resolution With Linear Probing (Extra)

datastructur.es

Open Addressing: An Alternate Disambiguation Strategy (Extra)

An alternate way to handle collisions is to use “open addressing”.

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 separate chaining.

datastructur.es
CC BY NC SA

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

datastructur.es