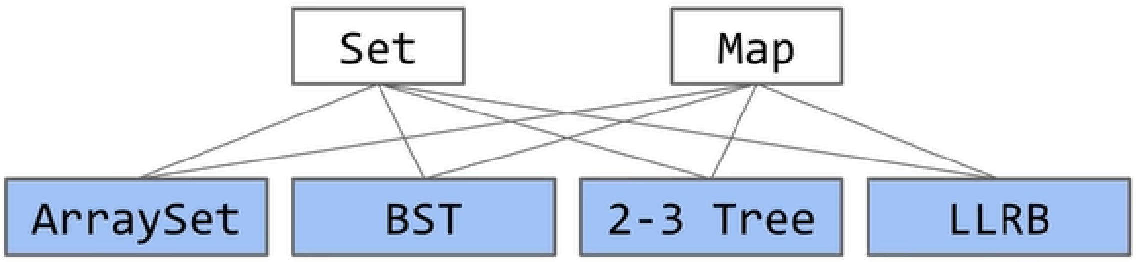
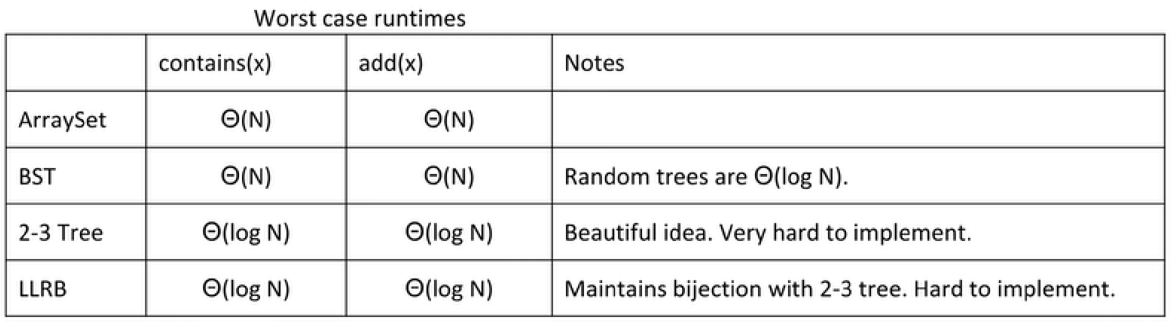
CSJ61B Lecture 19

Monday, March 30, 2020

Sets

We have 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.

Limits of Search Tree-Based Sets

Our search tree-based sets require items to 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 ,aybe be better?

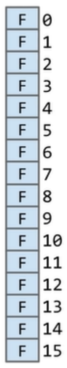
- Theta log N is amazing; 1 billion items is still only height ~30.
- Could we somehow do better than Theta log N?

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

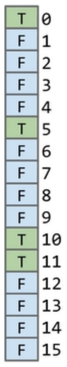
Using Data as an Index

One extreme approach: create an array of booleans indexed by data, where 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 nothing

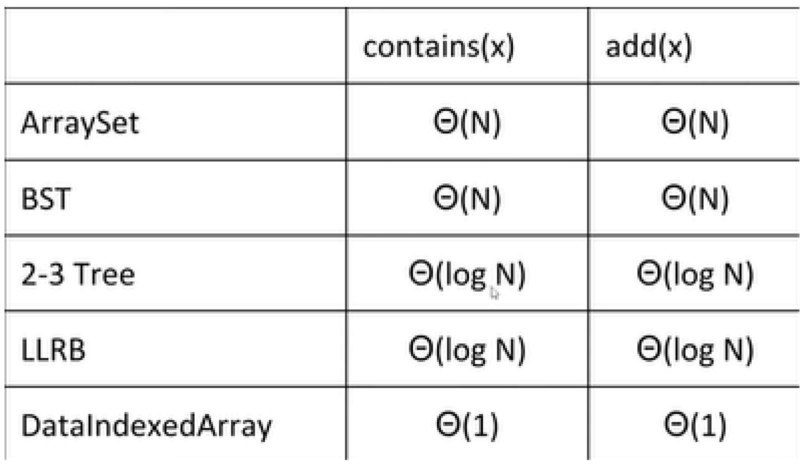


Set containing 0, 5, 10, 11

This method takes advantage of the fact that accessing the elements of an array are constant time no matter what.

Implementing `DataIndexedIntegerSet`

```
public class DataIndexedIntegerSet {
    private boolean[] present;
    public DataIndexedIntegerSet() {
        present = new boolean[2000000000];
    }
    public add(int i) {
        present[i] = true;
    }
    public 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)

We can see this whole thing is very fast! It's more of a proof of concept than a useful data structure really. But it sets us up for a nice introduction to hashing, where you see there are no actual comparisons happening.

There are downsides to this though: this is extremely wasteful of memory. To support checking presence of all positive integers, we need > 2 billion booleans, and we also need some way of generalizing beyond integers.

Data Indexed Word Set

Ideally, we want a data indexed set that can store arbitrary tyes. But for now, let's restrict ourselves to talking about `String`s, and we will get to generic types later.

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

Where do cat, bee and dog go in our array? What is the cat-th element of a list?

One idea is to use the first letter of the word as an index (a = 1, b = 2, ..., z = 26). This is a little dumb, because other words start with c, and we can't store strings like "=98yae".

Avoiding Collisions

Use all digits by multiplying each by a power of 27 (a = 1, b = 2, ..., z = 26). Thus, the index of "cat" is 3 x 27^2 + 1 x 27 + 20 x 27^0 = 2234.

Why this specific pattern? Well, in the decimal number system, we have 10 digits. If we want numbers larger than 9? Use a sequence of digits.

Example: 7091 in base 10

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

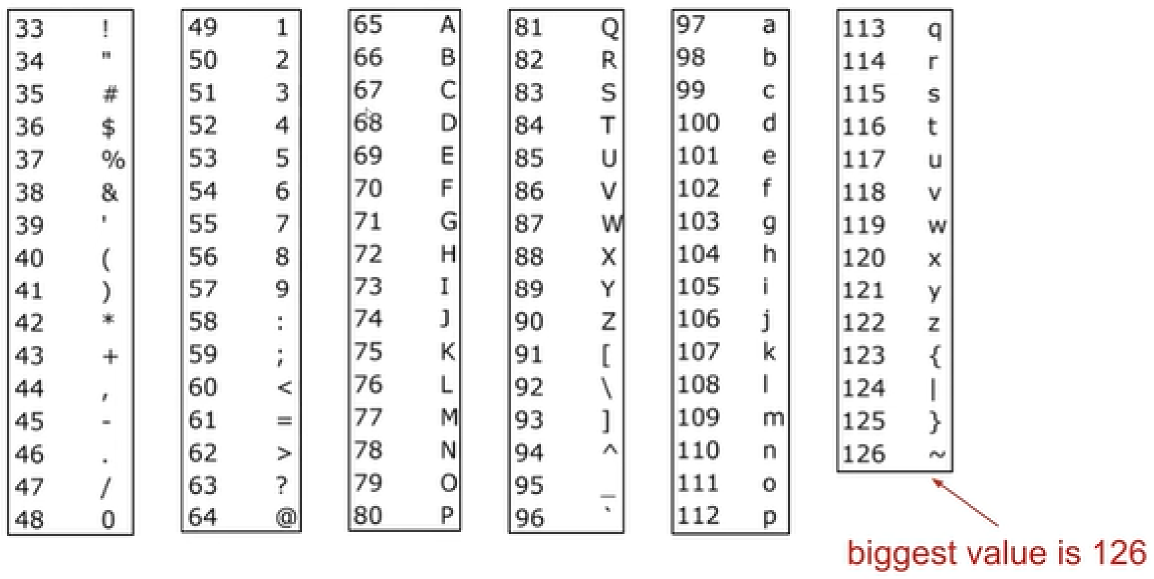
Thus, we are in effect sorting our numbers by a system in base 27. Test your understanding by converting the word "bee" into a number. The solution is 1598. As long as we pick a base greater than or equal to 26, this algorithm is guaranteed to give each lowercase English word a unique number! No other words will get the numbers 2234 and 1598, and we will never have a collision.

```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 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;
}
```

Generalizing Beyond Lowercase English

Using only lowercase English characters is too restrictive. What if we want to store strings like "2pac" or "eGg!"? To understand what value base we need, let's briefly discuss the ASCII standard. The most basic character set used by most computers is the ASCII format, where each character is assigned a value between 0 and 127. Characters 33 to 126 are "printable", and are shown below (the others include `\n` for newline).



33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o		
48	0	64	@	80	P	96	`	112	p		

biggest value is 126

`char c = 'D'` is equivalent to `char c = 68` using the table above. If we wanted to guarantee a unique value for all possible strings in the ASCII format, our function should use a base of 126 to ensure this.

As it turns out, because we no longer need to worry about uppercase characters, the implementation of `asciiToInt` is even simpler.

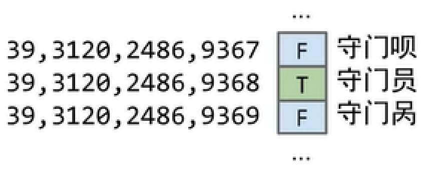
```java
public static int asciiToInt(String s) {
    int intRep = 0;
    for (int i = 0; i < s.length(); i += 1) {
        intRep = intRep * 126;
        intRep = intRep + s.charAt(i);
    }
    return intRep;
}
```

What if we wanted to go beyond ASCII? For example, in Chinese? Well, `char`s in Java also support character sets for other languages such as Chinese, with a standard known as Unicode, which is way too large to possibly list. Particularly, the largest possible value for Chinese characters is 40,959, so we would need to use this as a base if we want to have a unique representation of 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: 黎



Hash Codes

Integer Overflow

As it turns out, there is a biggest possible number in Java: 2,147,483,647. If you go over this limit, you overflow and start back over at the smallest integer, -2,147,483,648. This is important, as if we are trying to get integer representations of our strings, then we are going to run into overflow. The word omens in base 126 has a value of 28,196,917,171, and will overflow many times to produce -1,867-853,901.

Overflow can result in **collisions** too, causing incorrect answers:

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

The official term for the number we are computing is a "hash code", and it 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 is the set of Java integers, which is of size 4294967296, which means we are guaranteed to have collisions.

The 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 or strings, thus collusions are inevitable.

Thus, we have two fundamental challenges. One is how can we resolve hashcode collisions, otherwise known as collision handling? And the second is how do we even compute hashcodes for arbitrary objects? How does one quantify `Object`s other than strings?

Hash Tables

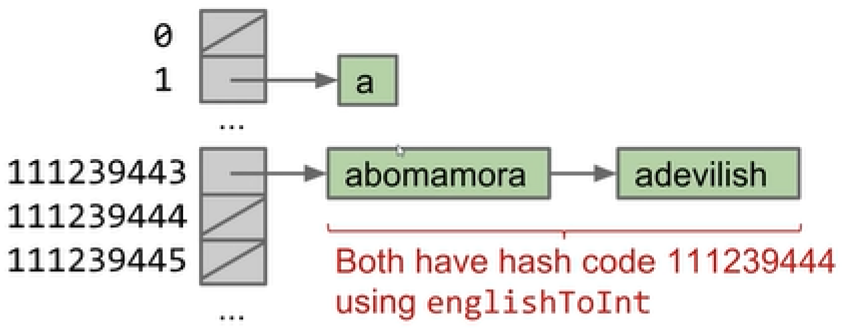
Given the inevitability of collisions, we are going to design a data structure that handles collisions well.

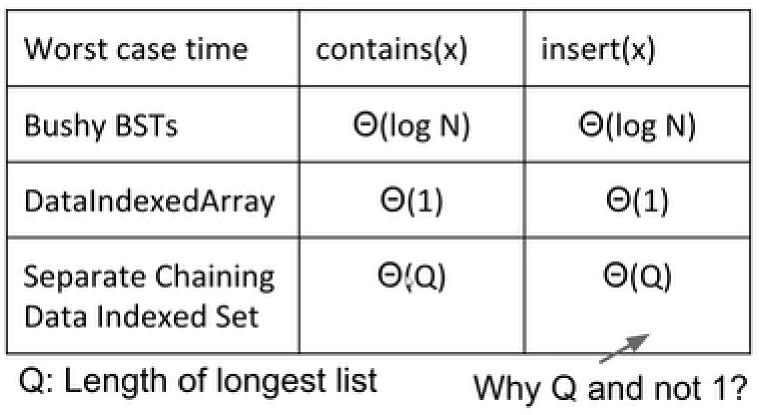
Here's an idea: rather than a boolean array, we should have an array of "bucket"s. We should store a "bucket" of items at position `h` according to their hashcode, at which point it would be relatively trivial to check the hopefully small number of items in the bucket. The implementation does not matter: we could use a `LinkedList`, `ArrayList`, or really any other data structure. For the purposes of this lecture, we will use a `LinkedList`.

Each bucket in our array is initially empty, and when we add an item `x` 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", because bucket # `h` is a "separate chain" of all items with hash code `h`.





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

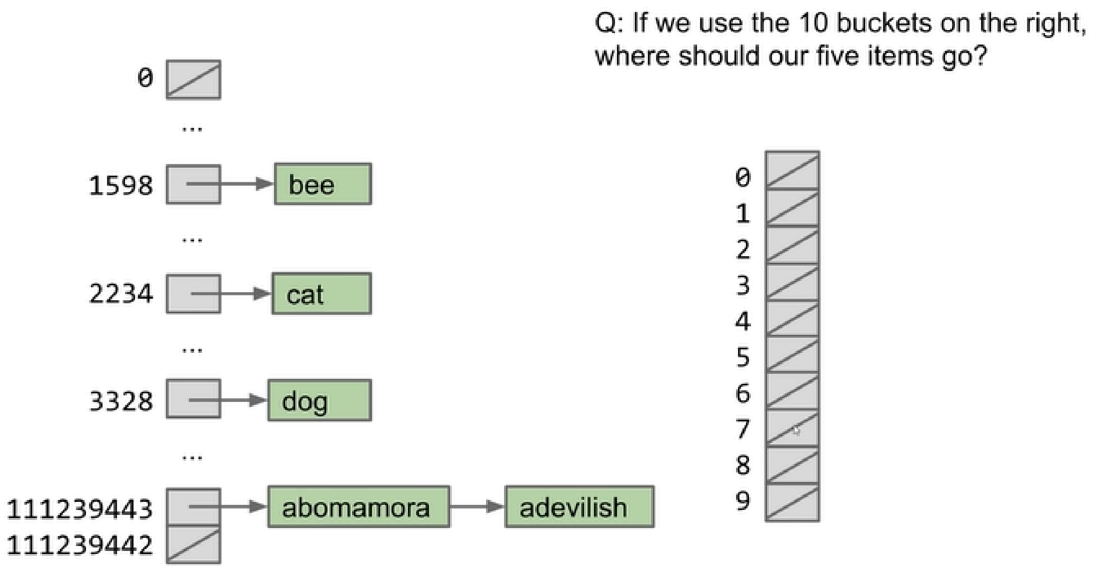
Q: Length of longest list

Why Q and not 1?

Why `Q` and not 1? Well, we aren't just sticking items into buckets: we need to check if they are already in the list, which is where the cost of the insertion operation comes from.

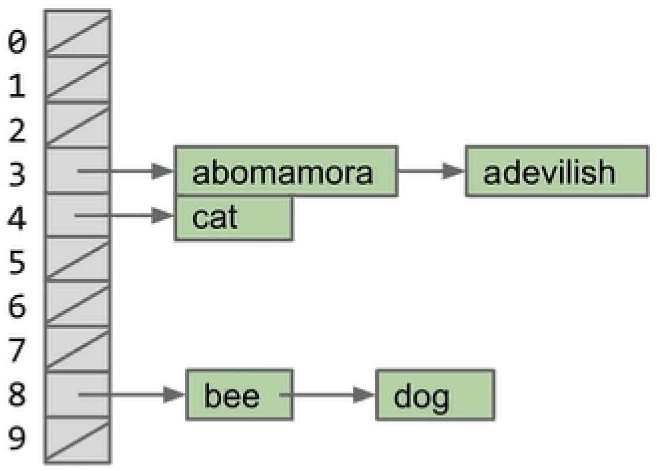
Saving Memory

Observe that we don't really need billions of buckets, which costs memory.



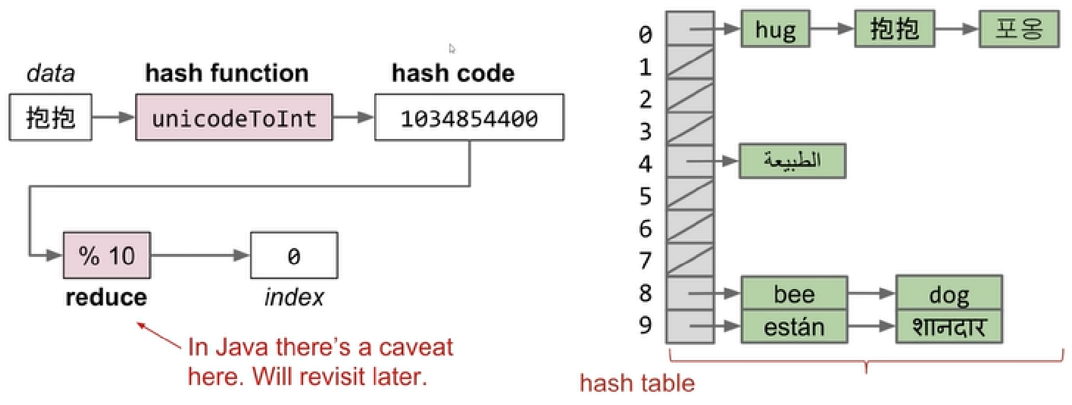
Q: If we use the 10 buckets on the right, where should our five items go?

One potential solution is to get the hash code and get its remainder when divided by 10.



reated is called a hash table:

- data is converted by a **hash function** into an integer representation called a **hash code**
- The **hash code** is reduced to a valid index, usually using the modulo operator.

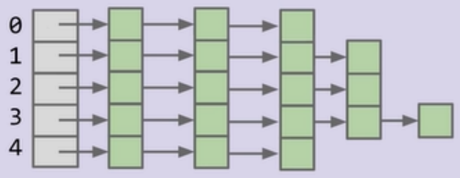


There is the caveat in modulo of how to handle negative hash codes which we will revisit.

Hash Table Performance

The good news with our hash table is that we use way less memory and can now handle arbitrary data as long we make it a hash function. The bad news is that the worst case runtime is now Theta Q, where Q is the length of the longest line.

Practice



Worst case runtimes

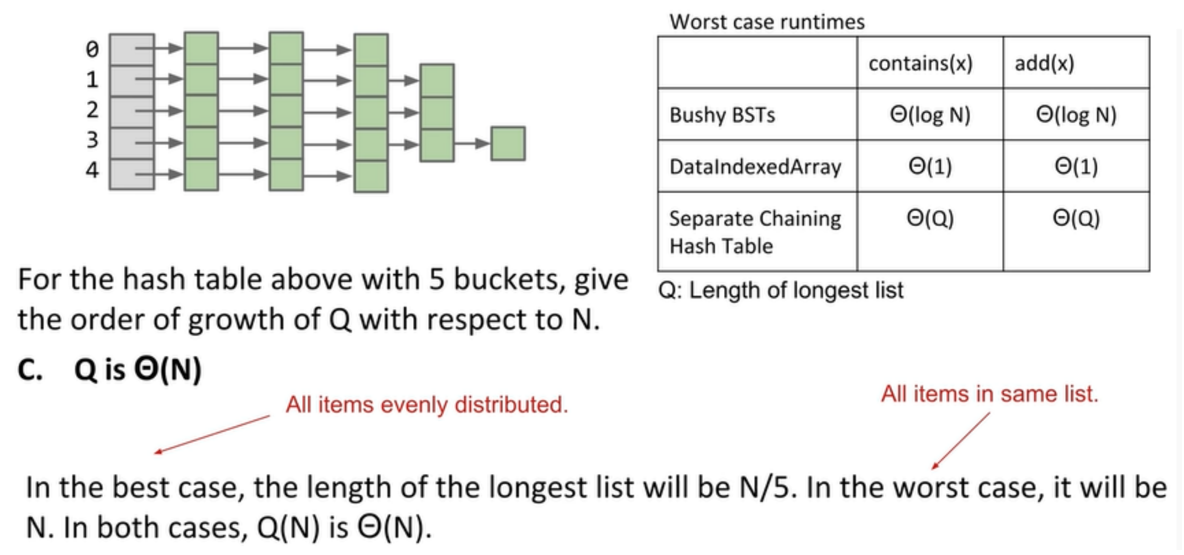
	contains(x)	add(x)
Bushy BSTs	Θ(log N)	Θ(log N)
DataIndexedArray	Θ(1)	Θ(1)
Separate Chaining Hash Table	Θ(Q)	Θ(Q)

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)

Solution



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.

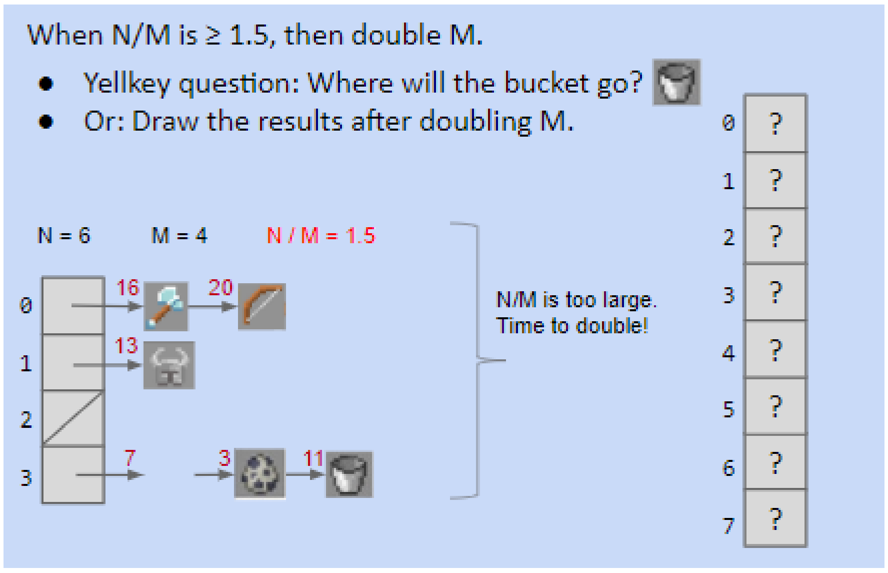
All items in same list.

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).

To improve the hash table, we should consider the case where we have a fixed number of buckets M, with an increasing number of items N. Our lists are currently spread out evenly, such that length Q = N/M. For M = 5, that means Q = Theta N, which results in linear time operations. How can we improve our design to guarantee a tight constant bound?

The solution is to also keep the number of buckets M increasing as well. We should keep track of the value N/M (the "load factor"), and if it exceeds some value, then we should resize M to be double.

Practice



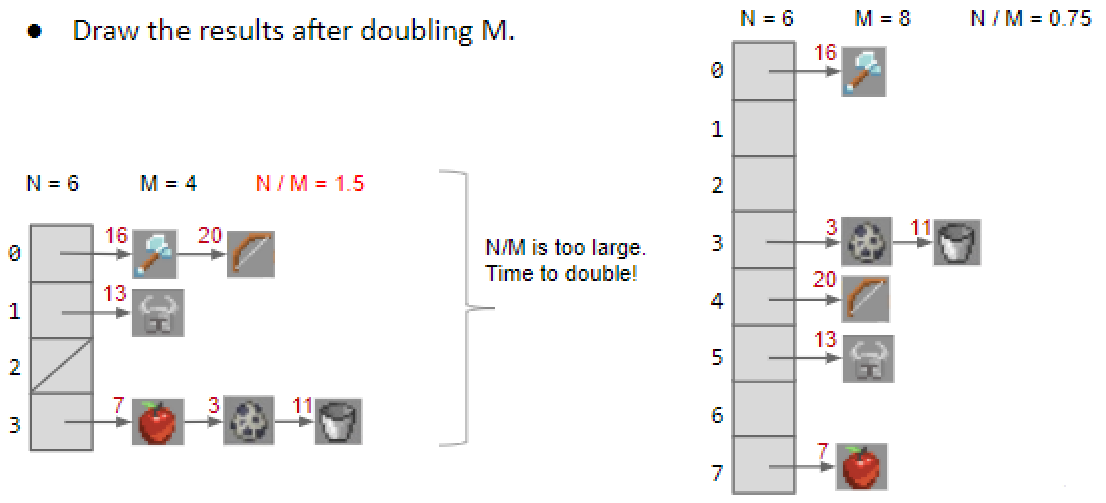
When N/M is ≥ 1.5, then double M.

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

Solution

When N/M is ≥ 1.5, then double M.

- Draw the results after doubling M.

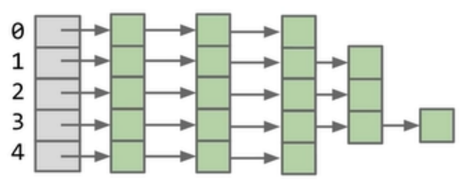


Resizing Performance

More importantly, we should ask how well the resizing operation performs. Well, as long as the number of buckets grows linearly, then O(N/M) will be constant overall.

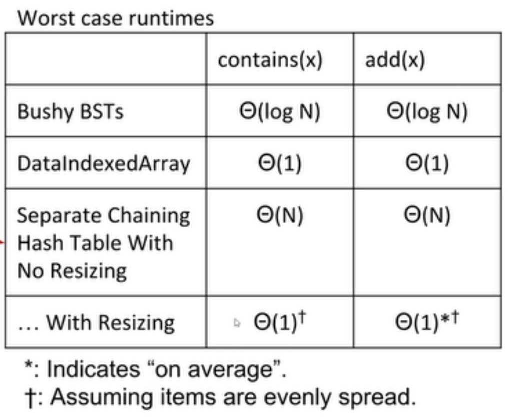
Assuming the items are evenly distribute like above, lists will be approximately N/M items long, resulting in Theta(N/M) runtimes.

Our doubling strategy ensures that N/M is constant, thus the worst case runtime for all operations is constant, unless that operation causes a resize. The resizing takes linear time, but since this is only performed sometimes, due to amortized cost, and as long as we resize by a multiplicative factor, the average runtime for the entire hash table will be constant.



Because Q = Θ(N)

Because Q = Θ(1)



Worst case runtimes

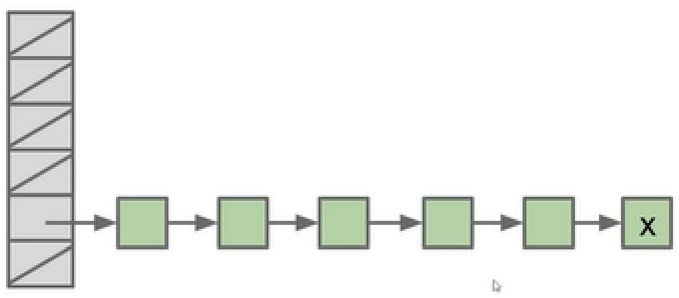
	contains(x)	add(x)
Bushy BSTs	Θ(log N)	Θ(log N)
DataIndexedArray	Θ(1)	Θ(1)
Separate Chaining Hash Table With No Resizing	Θ(N)	Θ(N)
… With Resizing	Θ(1)†	Θ(1)*†

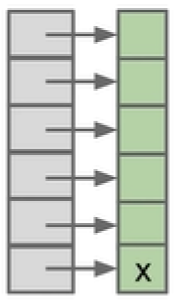
*: Indicates "on average".
†: Assuming items are evenly spread.

Hash table operations are on average constant time if:

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

An even distribution of items is critical for good hash table performance: Both of the below tables have a load factor of N/M = 1, but the left table will be much slower. How can we ensure even distribution? Well, to do so we need to discuss how hash tables work in Java.

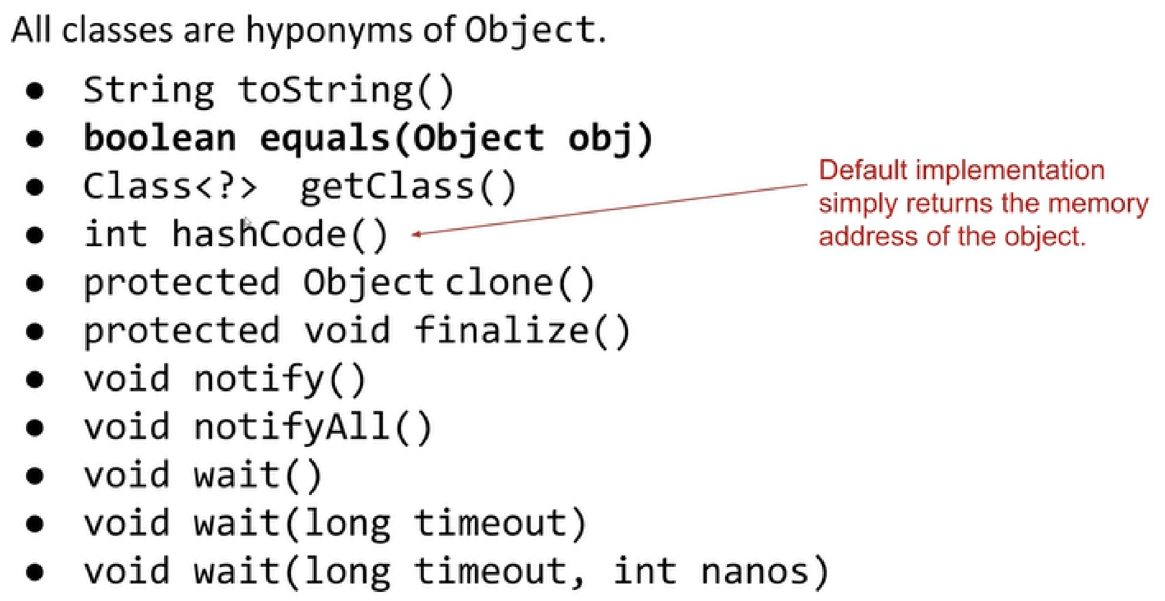




Java implementation

Hash tables are the most popular implementation for sets and maps. They have great performance in practice, don't require items to be comparable, and implementations are often relatively simple. Python dictionaries are just hash tables in disguise.

In Java, they are implemented as `java.util.HashMap` and `HashSet`. How does a `HashMap` know how tio compute each object's hash code? The good news is that it's not `implements Hashable`. Instead, all objects in Java must implement a `.hashCode()` method.



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)`

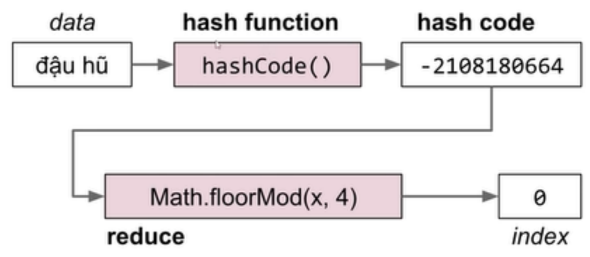
Default implementation simply returns the memory address of the object.

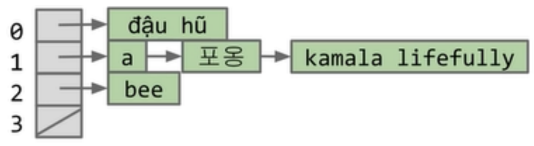
This means when you write a class, you can define how its `hashCode` is computed.

Handling negatives

Suppose some `Object`'s hash code is -1. Of the buckets 0, 1, 2 and 3, philosophically, which bucket is the most reasonable place to put this item?

1 and 3 are both reasonable answers, but for today, let's say 3, because -1 -> 3, 0 -> 0, 1 -> 1 etc. The reason we bring this up is that Java unfortunately doesn't do this: `-1 % 4 == -1`, which results in index errors! We should instead use `Math.floorMod(1, 4)`.





Warnings

There are two important things you must remember when using `HashMaps` and `HashSets`:

Never mutate objects in HashSet or HashMap

If an object's variables changes, then its hash code changes, and it may result in items getting lost.

Never override `equals` without also overriding `hashCode`

You should also never override the `equals` method without also overriding its `hashCode` method. This can lead to items getting lost as well, and generally weird behavior. `HashMaps` and `HashSets` use equals to determine if an item exists in a particular bucket.

Good Hash Codes

This topic is a little beyond the scope of our course, but let's discuss what makes a good hash code.

The real Java 8 hash code for strings has two major differences with our function from earlier:

- It uses base 31, because real hash codes don't care about uniqueness.
- And it stores (caches) calculated hash codes so future 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;
}
```

So which is better? Java's uses base 31, while ours is 126. Ignoring overflow, ours ensures a unique numerical representation for all ASCII strings. However, overflow is a particularly bad problem for base 126!

It has a major collision problem where:

- “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 will share the same `hashCode` because of overflow: 126^32 = 126^33 = 126^34 = ...0. The upper characters are all basically multiplied by zero and ignored.

Thus, a typical hash code base is a small prime. It avoids the overflow issue on the previous slide, and also reduces the chance of the resullting hash code having a bad relationship with the number of buckets. It is also small to reduce the cost of computation.

Summary and Miscallaneous

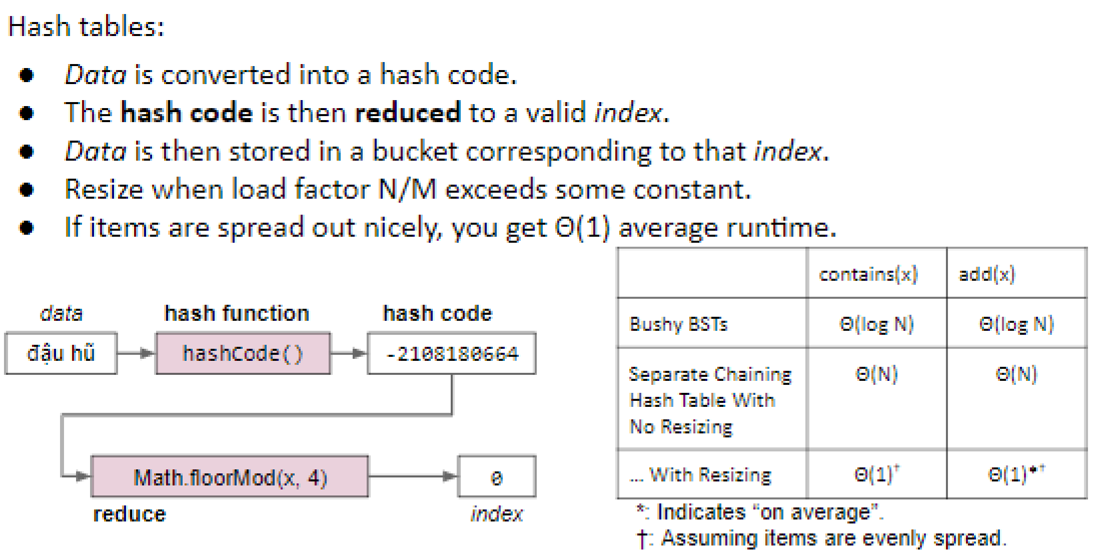
Lists are also like strings: a collection of items each with its own hash code.

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

We can save time by only looking at the first few items, which may marginally increase the chances of collision, but should not matter much. It will still work.

If we are computing the hash code of a recursive data structure, it will involve recursive computation. Here is how we compute the hash code of a binary tree.

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



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.