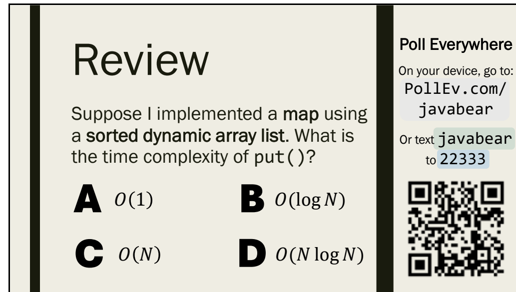
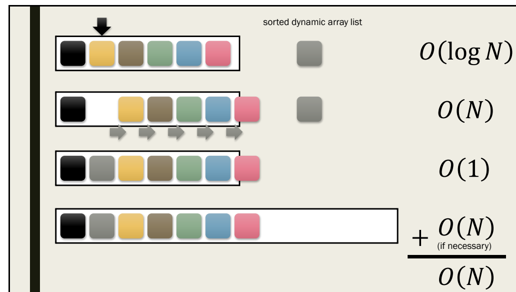
Slide 1



Slide 2



To put something into a sorted dynamic array list, first we can binary search for the position in logarithmic time. Then we need to shift every element over, doing a linear pass over the array as part of that process. Once space is available, inserting takes O(1) time. There is also the possibility that the backing array is full, and if so, we will need to resize the array, taking another linear pass to copy everything into the new, larger array. Either way, the total time is O(N) and the answer is C.

Slide 3

CS 2110

Lecture 20

Hashing

April 7th, 2026

Slide 4

Agenda

- Hash Tables
- Hash Functions
- Probing
- Chaining

Learning Outcomes

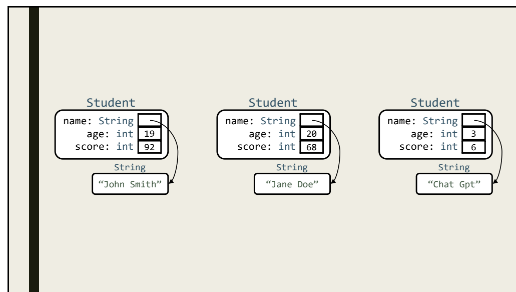
81. Compare the efficiency of Set implementations backed by arrays, sorted arrays, (balanced) binary search trees, and hash tables.
84. Identify the steps of hashing and describe the properties of a good hash function.
85. Visualize the state of a hash table after performing a given sequence of operations.
86. Describe the differences between hash tables that use chaining and probing for handling collisions. Identify scenarios where each implementation might be preferable.
87. Implement data structures backed by chaining or probing hash tables.

Slide 5

HASH TABLES

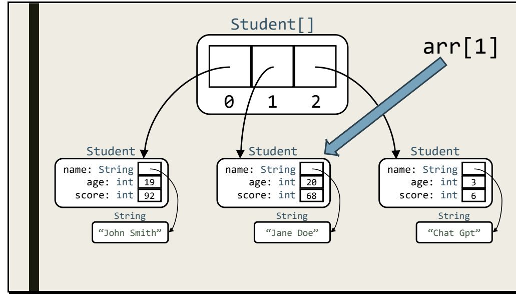
Today we're talking about what I personally consider the most broadly useful, magical O(1) everything data structure: the hash table.

Slide 6



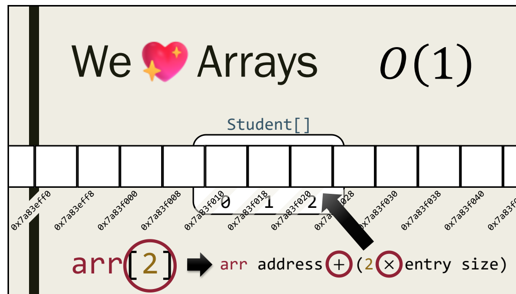
Imagine we needed to store a large number of Student objects in a data structure, but that all our students were numbered 0, 1, 2, 3, etc.

Slide 7



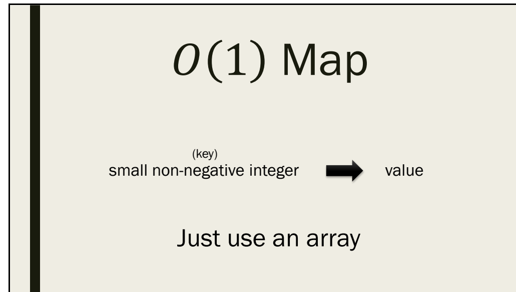
There is an obvious data structure we already know that can store these: an array. If we need to find student number 1, we just write index into the array and are able to get that student immediately out.

Slide 8



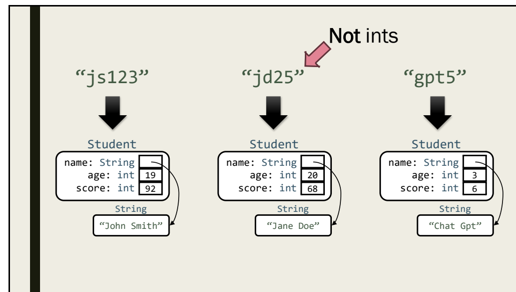
This is able to happen in O(1) time because, at its core, an array is just a contiguous section of memory. Your computer's memory is just a bunch of memory locations, each numbered sequentially. When we write something like `arr[4]`, the compiler turns this into the address of the array plus the index times the size of each entry. Notice that this is just pure integer arithmetic, which can be performed in O(1) time. The result is the memory address of the array entry itself, which can be accessed directly. There is something inherent to integers that allows us to use them to index into memory in O(1) time, as memory addresses are ultimately just integers themselves.

Slide 9



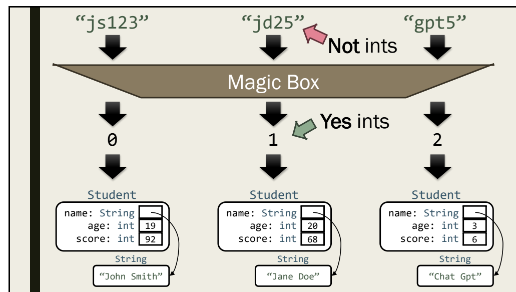
This means if you need a map from small non-negative integers as the key to any value, just use an array. It's the original O(1) map.

Slide 10



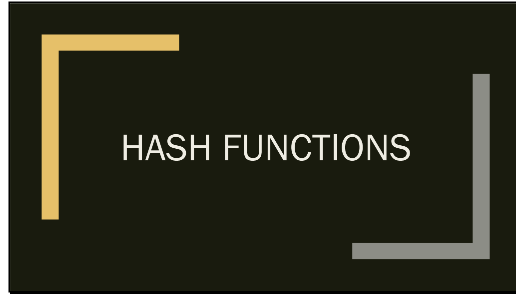
Unfortunately, we often don't want to use integers as our keys. With students, for example, we often want to look for them by net ID, which are strings. This means we can't use the array trick to find a student in O(1) time…

Slide 11



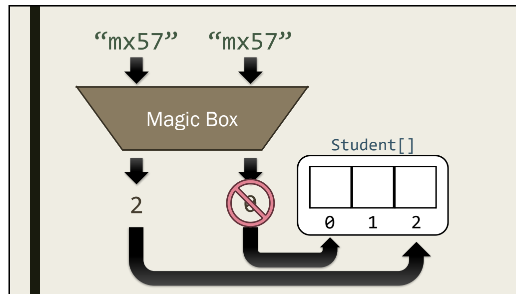
… unless we had some magic box that could convert our keys, which are arbitrary Java objects (and not ints), into an int. If that were the case, then we could just use an array again, using the int to index just like we did earlier. So the only question is to find this magic box.

Slide 12



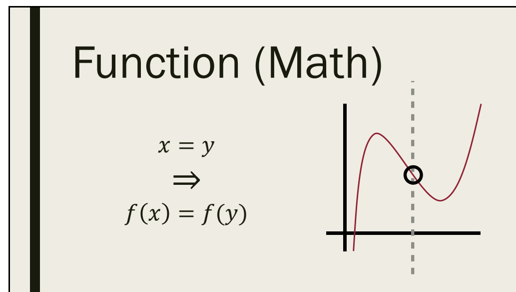
To do so, we're going to steal some ideas from math.

Slide 13



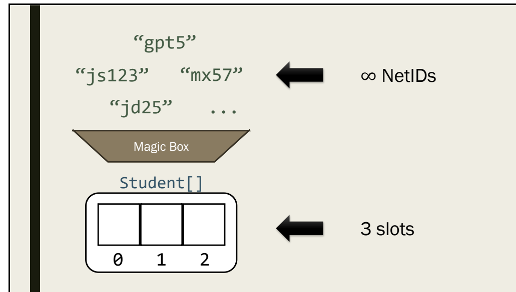
The first and most important property we need is that if we feed the same net ID into the box, it needs to give us the same int every time. Otherwise, if I stick my student record in index 2 at first, and then later look in index 0, I'm not going to be able to find my record.

Slide 14



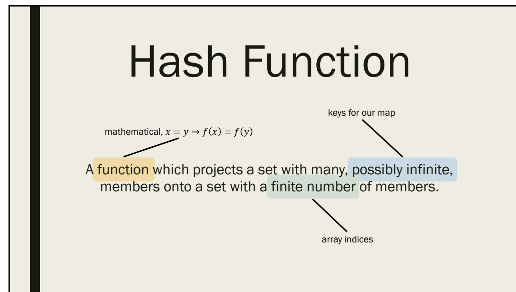
This is the mathematical notion of a **function**, where the same input must provide the same output. You may remember doing the vertical line test back in middle school; this is what that is.

Slide 15



The other main property is that there are possibly infinite net IDs but only 3 slots in my array. So this magic box needs to take a possibly infinite input domain and output only a discrete, finite number of possible results.

Slide 16



Mathematicians have a name for this – they call it a hash function.

Slide 17

Java’s `hashCode()`

- Method in `Object` class
- By default, guaranteed equal for same object in memory
- **Invariant:** `x.equals(y)` ⇒ `x.hashCode() == y.hashCode()`

```
public class Object {
  public int hashCode();
}
```

This is the idea we’re going to steal and bring back to our programming language. It turns out Java provides a built-in hash function on every object by default via the `hashCode()` method in the `Object` class. Since every class is a subtype of `Object`, every object automatically gets a built-in hash function. Note that if you override the `equals()` method, you must also override the `hashCode()` method to ensure they agree – this is that mathematical notion that if two objects are equal, then running the hash function on both must return the same result.

Slide 18

```
public class Object {
  public int hashCode();
}
```

any int (even negative ones)

Object → hashCode() → -437853248 → abs(h % len) → 2

Pre-Hashing | Hash Code | Indexing | Hash Value

Unfortunately, the built in `hashCode()` method can return any Java `int`, which isn't particularly useful if we're trying to index into an array. So we have to modulo the length of the array and then take the absolute value.

Slide 19

"jd25" "mx57"

hashCode() Magic Box abs(h % len)

1 2

Student[]

0 1 2

Student

name: String

age: int 20

score: int 68

String

"Jane Doe"

put(key, value)
1. Hash the key
2. Put value in resulting index

get(key)
1. Hash the key
2. Get value from index if present

So that's our magic box. We use the built in `hashCode()` method followed by a modulo and absolute value to convert any key into an int. Then we use that int to index into an array. Assuming the hash function runs in O(1) time, we have now built an O(1) map.

Slide 20

Poll

$hashCode(x) = \left|\frac{1}{x}\right|$

½

0 1 2 3 4 5

Given a **mutable** `Fraction` key of ¼, if I mutate it to ½ when it's inside the hash table, what happens when we try to find the key ½?

A It's fine (hash code updated and is now 2)

B Faster to find (earlier in table)

C Can no longer find (in wrong place)

D Slower to find (checking more spots)

Poll Everywhere

On your device, go to: PollEv.com/javabear

Or text javabear to 22333

Slide 21

Mutability Tangent

- If key mutates, its hash code may be wrong, rendering the key unable to be found
- **Do not use mutable keys** in a hash table (mutable values are okay)

Slide 22

Hash Table

1. Use `hashCode()` to turn input into `int`
2. Use `int` to index into array

To recap, this is the idea of a hash table. It allows us to implement the Map ADT by using the idea of a hash function to turn our keys into integers, which we can use to index into arrays.

Slide 23

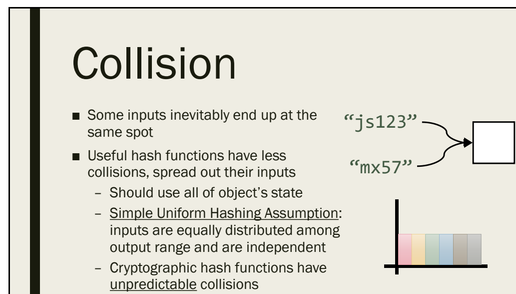
Injectivity

"gpt5"
"js123"
"mx57"
"jd25"
...
∞
?
insert pigeonhole joke here

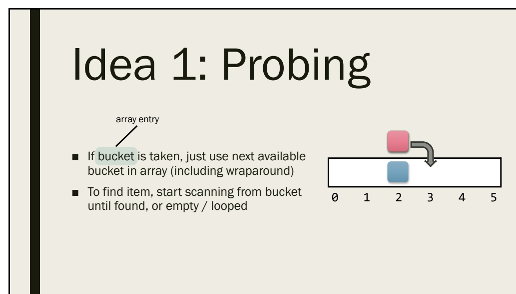
Except for one final problem. By definition, a hash function takes a large, possibly infinite domain of inputs to a finite number of output values. Thus, by the pigeonhole principle, eventually, we're going to run out of unique outputs. When this happens, we have…

Slide 24

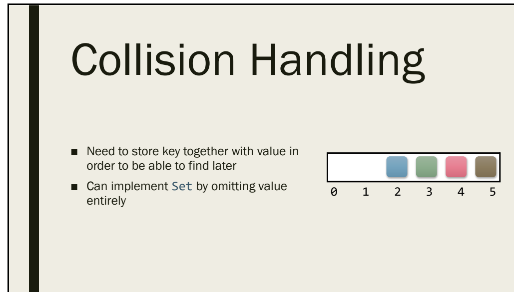
Slide 25



Slide 26



Slide 27



Notice that in order to be able to ensure we've found the correct value for any given key during the probing process, we now are forced to store the key together with the value inside our array. But as a benefit, this means a hash table can also be used to implement a Set ADT, not just a Map, by omitting the value and only storing the element we're indexing with.

Slide 28

Probing Example

Map from Net ID (string) to score (int)

hashCode(netID) = numeric portion of net ID

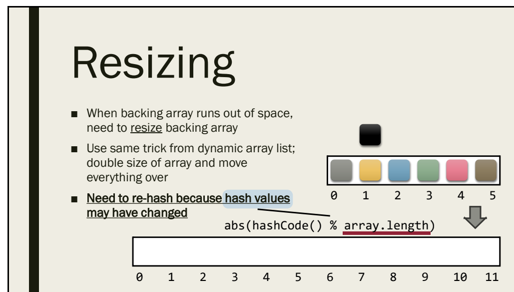
0	1	2	3	4
K:"mlx29" V:100	K:"bob10" V:86	K:"aaa20" V:99		K:"dx4" V:73

```
put("dx4", 73)
put("mlx29", 100)
put("bob10", 86)
put("aaa20", 99)
```

```
get("bob10")
get("gpt4")
```

This is a full example of what a probing hash table implementing a Map might look like. Note that getting "gpt4" would return nothing, as the probe stops when it reaches the first empty bucket (or when it loops all the way around, in the case the array is full).

Slide 29



Slide 30

Resize Poll

After resizing, where should 1/7 end up?

$hashCode(x) = \left|\frac{1}{x}\right|$

2/7	1/6	1/2	1/3	1/7
0	1	2	3	4

0 1 2 3 4 5 6 7 8 9

A B C D

Poll Everywhere

On your device, go to: PollEv.com/javabear

Or text javabear to 22333

Slide 31

Load Factor

- Measure of how "full" your hash table is
- Performance degrades when approaching 1
- Need to resize when gets large
  - Don't need to wait until 1
  - Java resizes at 0.75

$$\lambda = \frac{\#\ of\ elements}{\#\ of\ buckets} = \frac{N}{C}$$

Capacity: total number of buckets (length of array)

Slide 32

Time Complexity

- Resize if necessary — $O(N)$, $O(1)$ amortized
- Run `hashCode()` (hard) — $O(1)$ w.r.t. hash table
- Compute hash value (% abs) — $O(1)$
- Probe for position — $\sim O(1)$ in expectation
- Access array index directly — $O(1)$ (Assuming Low Load Factor)

(expected) $\sim O(1)$ amortized

Slide 33

Remove Poll

Given this hash table, what happens if I remove 1/3?

0	1	2	3	4	5
		½	⅓	3/7	

$hashCode(x) = \left\lfloor \frac{1}{x} \right\rfloor$

A Cannot find 1/2

B Cannot find 3/7

C Cannot find both

D It's fine (let's be real you know it's not this)

Poll Everywhere

On your device, go to: PollEv.com/javabear

Or text javabear to 22333

Slide 34

Tombstones

- Probing relies on continuous sequence of elements being present
- Removing elements breaks the sequence
- Leave tombstone element in place when removing
  - Tombstones count in load factor, but can be overridden by new elements and should **not** be copied during rehashing

0	1	2	3	4	5
		½		3/7	

Slide 35

Probing

Pros

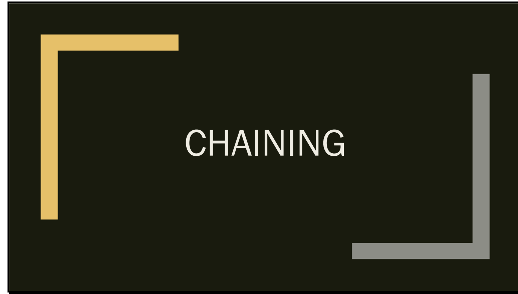
- Everything in same array (fast cache lookup; see CS 3410)
- Memory efficient (if mostly adding)

Cons

- Sensitive to clustering (needs *very* good hash function to avoid too many collisions)
- Tombstones

If the blue and green elements hashed to the same index, the green would be probed to the next spot over. Now, if the red element hashes to that next index up, it has collided with the green element even though there was no collision to begin with in the original hash function. This is clustering, when values are close together even if they don't directly collide. Probing hash tables do not handle clustering very gracefully.

Slide 36



Slide 37

Idea 2: Chaining

- Stick collisions in a linked list
- That's it

Each array entry is now a linked list

Slide 38

Load Factor

What is the max load factor of each type of hash table?

Probing \ Chaining	1	∞
1	A	B
∞	C	D

Poll Everywhere

On your device, go to: PollEv.com/javabear

Or text javabear to 22333

Since probing keeps everything inside the array, it can only store as many elements as the number of buckets in the array. Chaining, on the other hand, has no fixed limit, since each linked list can grow as long as it needs.

Slide 39

Time Complexity

- Resize (if wanted) — $O(N)$, $O(1)$ amortized
- Run `hashCode()` — $\sim O(1)$ hopefully
- Compute hash value (% abs) — $O(1)$
- Traverse list — $\sim O(\lambda)$ in expectation (Simple Uniform Hashing Assumption + Low Load Factor)
- Access array index directly — $O(1)$

(expected) $\sim O(1)$ amortized

Since chaining allows for load factors as high as we want, resizing is not strictly required at any point. However, to keep the runtime complexity O(1), we must keep the load factor low since the runtime of traversing the list depends on the load factor. Thus, resizing is still done when the load factor grows large.

Slide 40

Chaining

- More resilient to bad hash functions
- Java's default choice
- No longer localized to one array

Slide 41

Other Ideas (not in scope for exams)

- Probing
  - Use second hash function to determine stride
  - Cuckoo Hashing: check two positions; displace if both occupied and repeat
- Chaining
  - Chain with something better than a linked list

Slide 42

Recap

- Arrays give $O(1)$ access
- Can use hash function to turn key into index into array to build $O(1)$ sets and maps
- Resize array and rehash when load factor is too high
- Collisions can be resolved by probing or chaining
  - Probing is sensitive to clustering, requires tombstones for removal
  - Chaining uses more memory, is more resilient

Slide 43