Programming Abstractions

CS106B

Cynthia Lee, Julie Zelenski, Neel Kishnani

Today’s Agenda

- Analyzing ADT Implementations

- Implementing ADTs so far
  - Arrays
  - Binary Search Trees

- Hash tables
  - Hash functions
  - What makes a “good” hash function?

- Other uses of hashing

Analyzing ADT Implementations

Analyzing ADT Implementations

Our goal is to achieve fast

- Contains 🔎
- Add 🗃️
- Remove 🗑️

Review: Implementing ADTs so far

🛠️

Implementing Set

- Let's use an array!

- We need dynamic memory (on the heap!)

- 2 versions: unsorted array and sorted array

Unsorted Array

Need to check if the element is contained in the `Set`

Contains

Add

Remove

Unsorted Array

Need to check if the element is contained in the `Set`

Contains	O(n)
Add	
Remove	

Unsorted Array

Need to check if the element is contained in the `Set`

Contains	O(n)
Add	O(n)
Remove	

Unsorted Array

Need to check if the element is contained in the `Set`

Contains	O(n)
Add	O(n)
Remove	O(n)

Sorted Array

Binary search speeds up lookups!

- Contains
- Add
- Remove

Sorted Array

Binary search speeds up lookups!

Contains	O(log(n))
Add	
Remove	

Sorted Array

Still need to shift elements over 😕

Contains	O(log(n))
Add	O(n)
Remove	

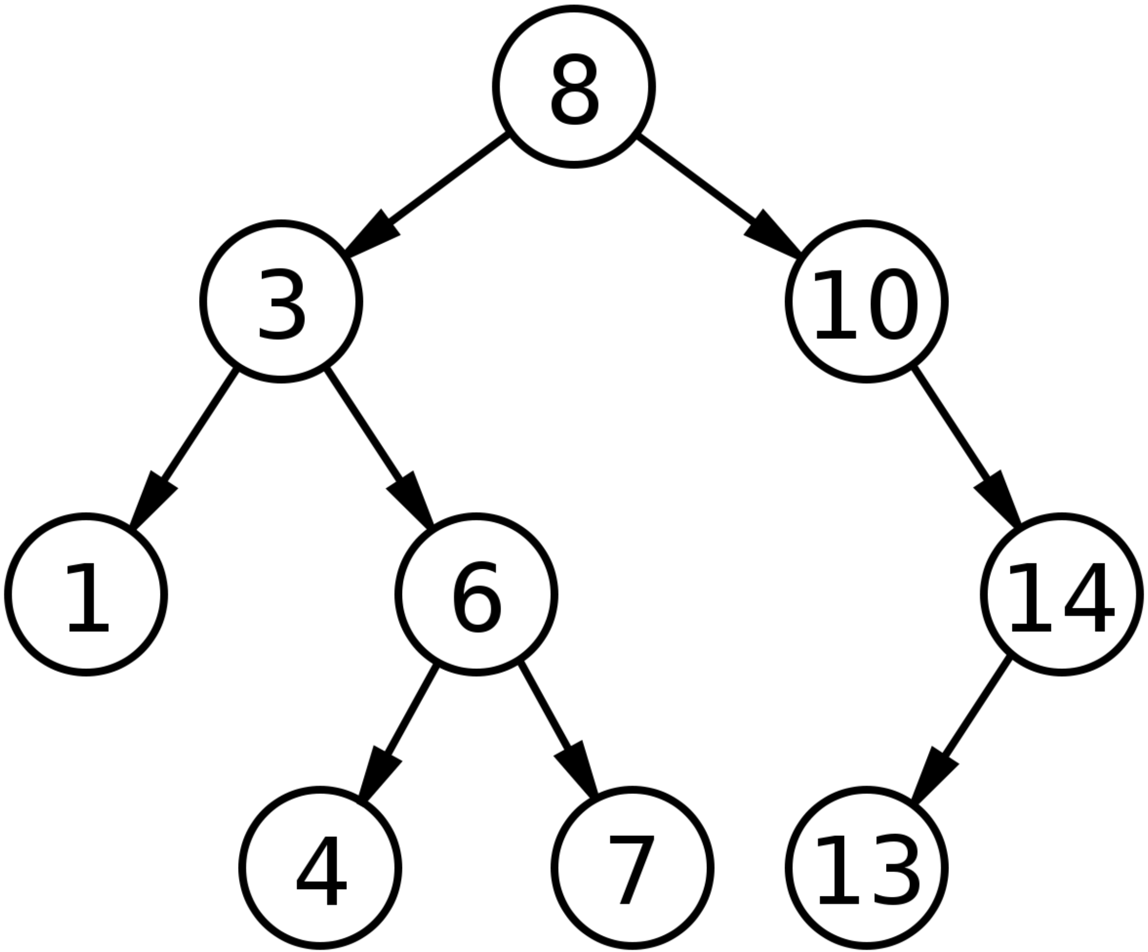
Sorted Array

Still need to shift elements over 😕

Contains	O(log(n))
Add	O(n)
Remove	O(n)

Next step for lookup-based structures...

Binary Search Trees 🌳


8
3
10
1
6
14
4
7
13

Stanford library `Map` and `Set` classes are backed by binary search trees

Binary Search Trees

Assuming a balanced binary search tree

Contains

Add

Remove

Binary Search Trees

Assuming a balanced binary search tree

Contains	O(log(n))
Add	
Remove	

Binary Search Trees

Assuming a balanced binary search tree

Contains	O(log(n))
Add	O(log(n))
Remove	

Binary Search Trees

Assuming a balanced binary search tree

Contains	O(log(n))
Add	O(log(n))
Remove	O(log(n))

Can we do better than `O(log(n))`? 🤔

Some context before answering that question


Student Package Center
UG2 Mail & Logistics
STANFORD HAIR

UG2 Package Center

- The package center gets a lot of packages throughout the quarter

- They store packages by keeping a small number of buckets for groups of packages

UG2 Package Center

- They have a rule that assigns packages to buckets

- When a student comes in to pick up their package, they know exactly which bucket to go to

To: Neel Kishnani

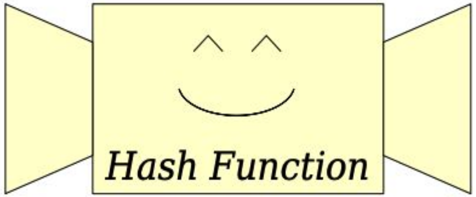
Unique ID: NEELK

Bin Number: G-B1A1

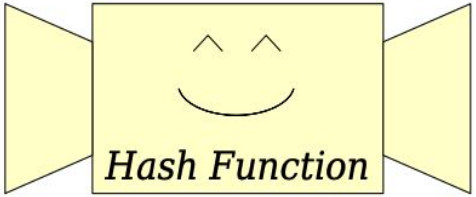
11/15/2021 4:19 PM

JJD014600009239261945

Let's introduce a special function called a hash function



We'll use this hash function to assign elements to buckets



Hash Functions

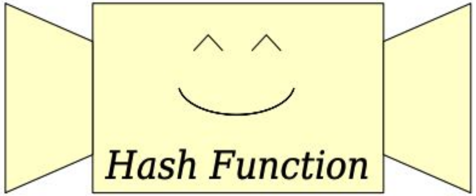
Important property:

The same input should produce the same output

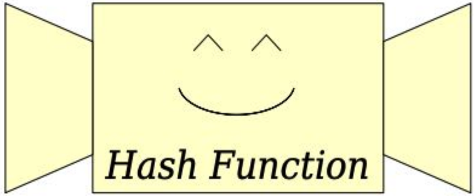
- Functions with this property are deterministic
- More on deterministic functions in CS103!

For the purposes of CS106B, assume our hash function returns an `int`

The input can be of any type though! (`string`, `double`, `int`, etc.)



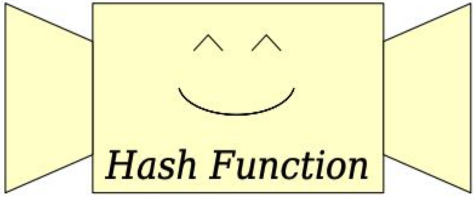
Input: 12



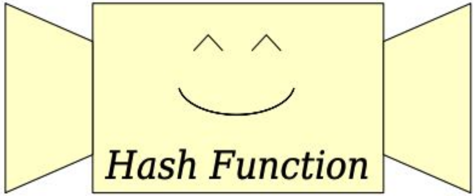
Input: 12

Hash Code: 106107

The output of a hash function is called a hash code!

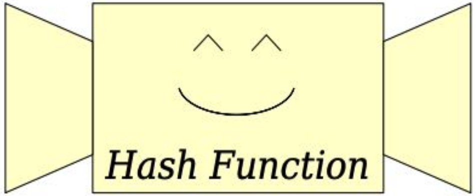


Input: 137

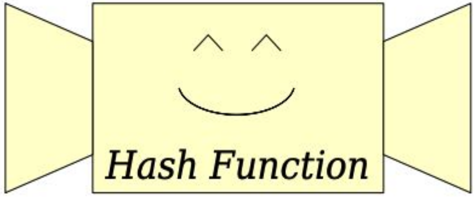


Input: 137

Hash Code: 309731



Input: 12



Input: 12

Hash Code: 106107

A new data structure 🪣

- Let’s go back to our array and treat each slot as a bucket for elements, just like the package center!

- We’ll assign each element we need to insert into a bucket and store it there

Use a hash function to assign elements to buckets 🪣

This data structure is called a

Hash Table

```
HashTable::HashTable() {
    // Initialize array of buckets
    _elements = new int[NUM_BUCKETS];
}
```

An idea for a hash function

Return the element itself!

```
int hash1(int elem) {
    return elem;
}
```

```
void HashTable::insert(int elem) {
    int bucket = hash1(elem);
    _elements[bucket] = elem;
}
```

Break

Logistics

- Assignment 6 grace period ends tonight (11/19) at 11:59PM

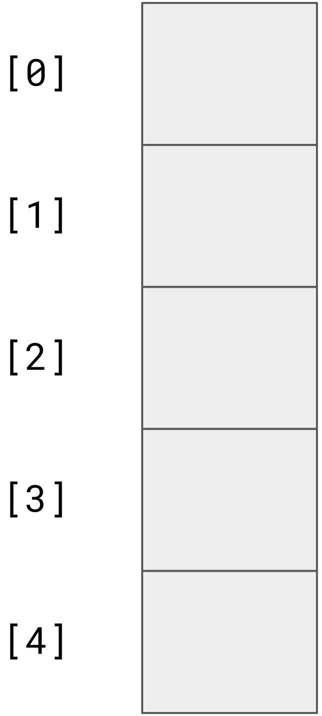
- Assignment 7 is out now and due 12/1
  - Huffman Coding!
  - Assignment 7 YEAH is today at 2:30PM in Hewlett 201

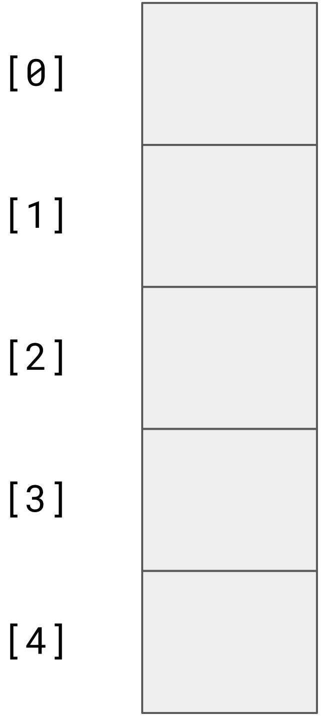
Logistics

- Final Diagnostic:
  - 24 hour window on **Monday December 6th**
  - Same format as midterm
  - Stay tuned for practice materials

Resume

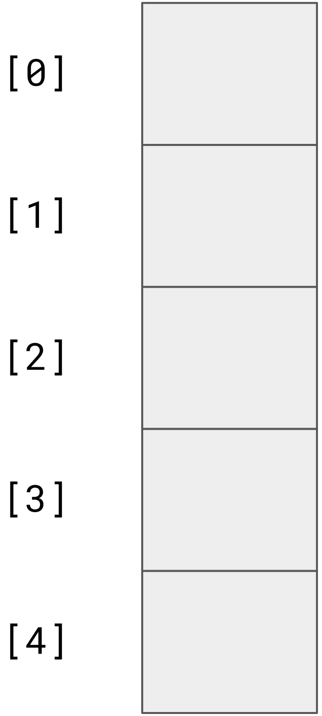
Our Buckets





Hash Function:

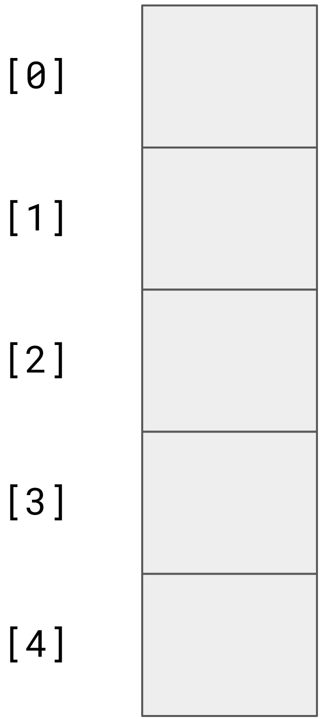
```
int hash1(int elem) {
    return elem;
}
```



Hash Function:

```
int hash1(int elem) {
    return elem;
}
```

Input: 3

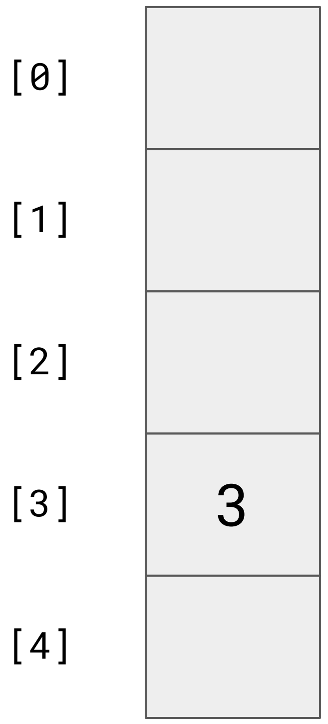


Hash Function:

```
int hash1(int elem) {
    return elem;
}
```

Input: 3

Hash Code: 3



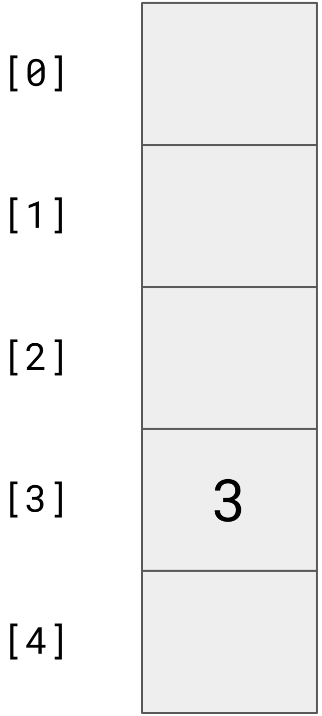
Hash Function:

```
int hash1(int elem) {
    return elem;
}
```

Input: 3

Hash Code: 3

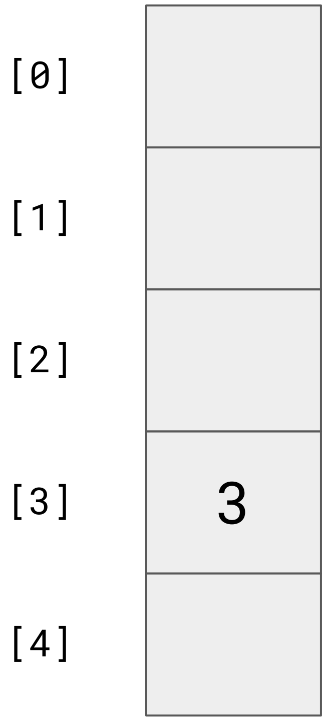
The hash code is the bucket we put the element in



Hash Function:

```
int hash1(int elem) {
    return elem;
}
```

Input: 0

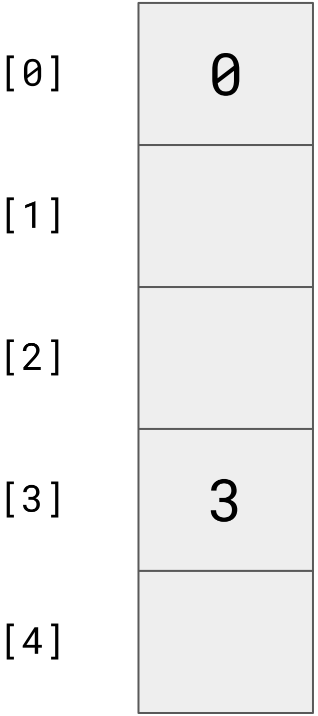


Hash Function:

```
int hash1(int elem) {
    return elem;
}
```

Input: 0

Hash Code: 0

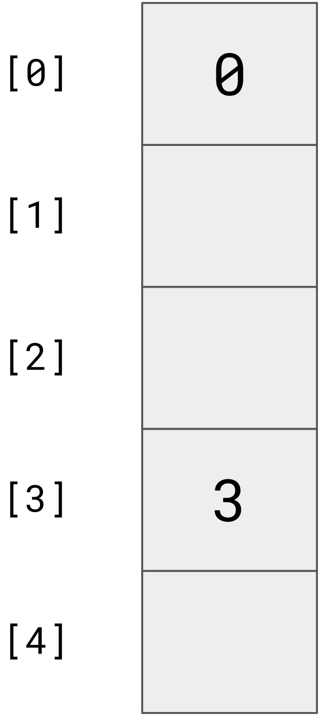


Hash Function:

```
int hash1(int elem) {
    return elem;
}
```

Input: 0

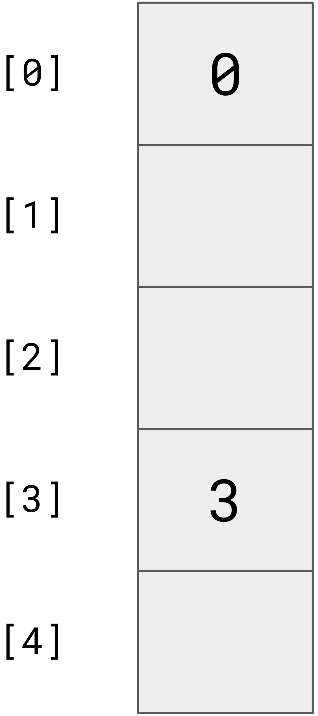
Hash Code: 0



Hash Function:

```
int hash1(int elem) {
    return elem;
}
```

Input: 17000

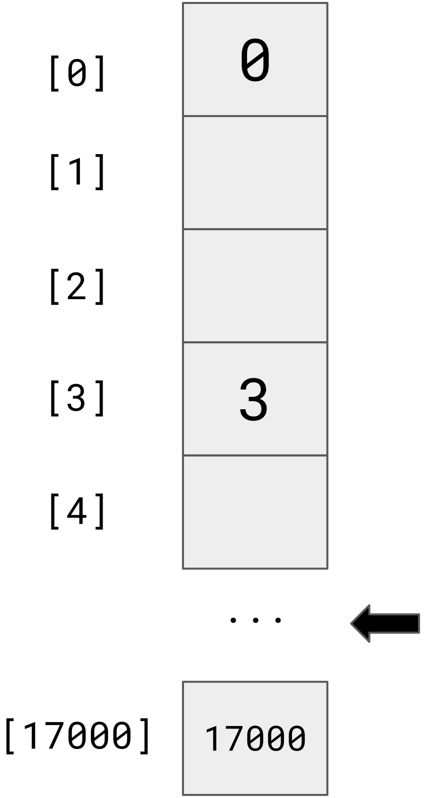


Hash Function:

```
int hash1(int elem) {
    return elem;
}
```

Input: 17000

Hash Code: 17000



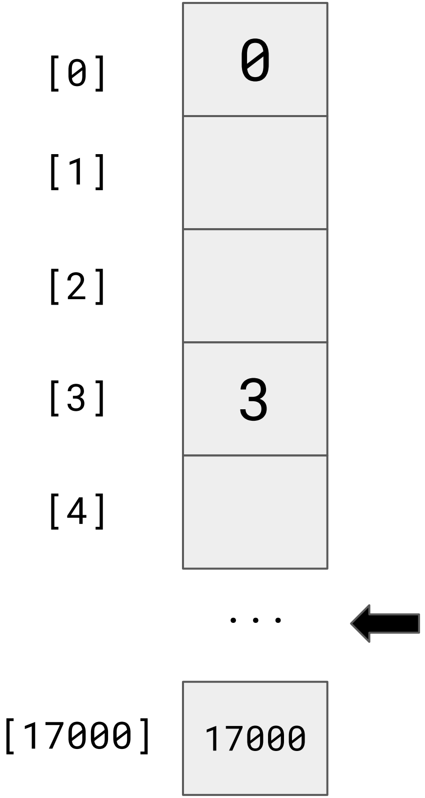
We need to enlarge our array

Hash Function:

```
int hash1(int elem) {
    return elem;
}
```

Input: 17000

Hash Code: 17000



Lots of wasted space here!

Hash Function:

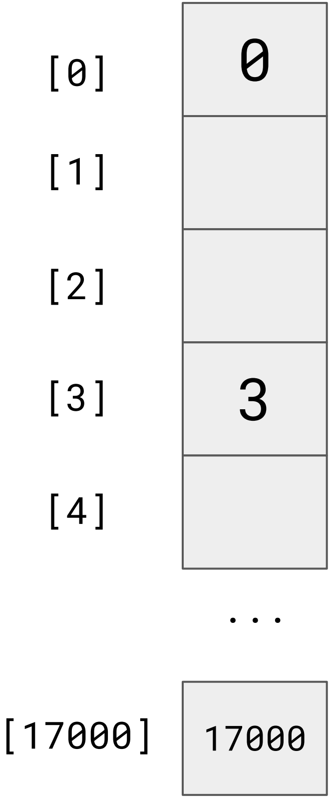
```
int hash1(int elem) {
    return elem;
}
```

Input: 17000

Hash Code: 17000

Issue #1

This hash function could lead to a sparse hash table



Hash Function:

```
int hash1(int elem) {
    return elem;
}
```

Input: -3

Issue #2

This hash function doesn’t handle negative inputs

Issue #3

We don’t initialize the buckets, so there’s a chance that an “empty” bucket could have a value

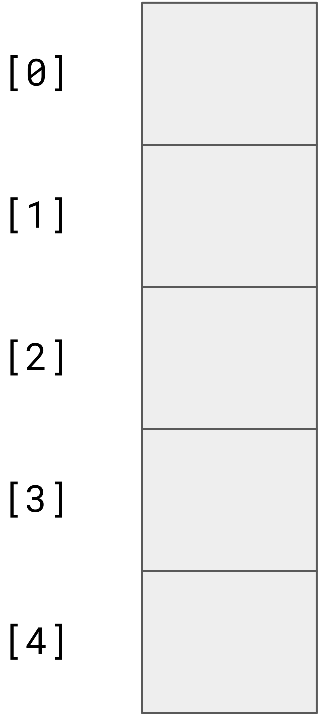
(i.e. bucket N could have N in it as a “garbage” value leading to an incorrect check on `contains`)

We want to limit the range of possible buckets

A better(?) hash function 💭

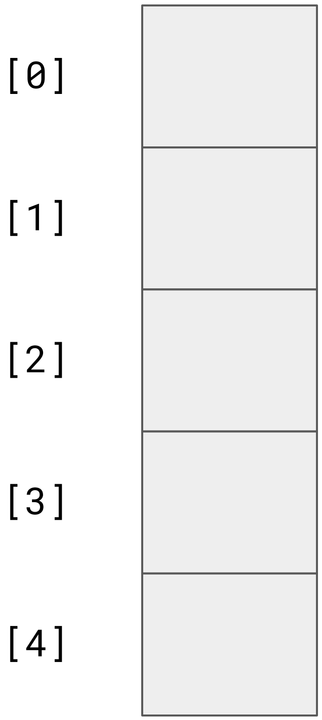
Let's use the % operator!

```
int hash2(int elem) {
    return abs(elem) % numBuckets;
}
```



Hash Function:

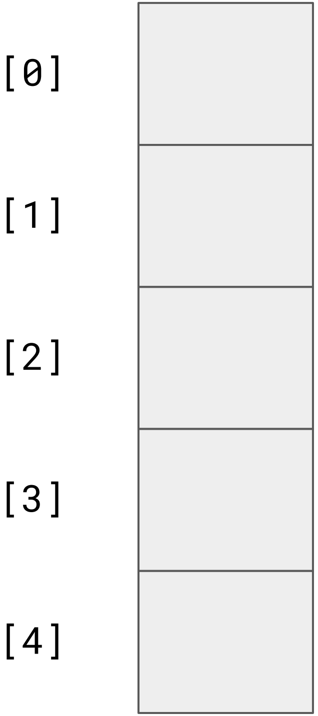
```
int hash2(int elem) {
    return abs(elem) % numBuckets;
}
```



Hash Function:

```
int hash2(int elem) {
    return abs(elem) % numBuckets;
}
```

Input: 3

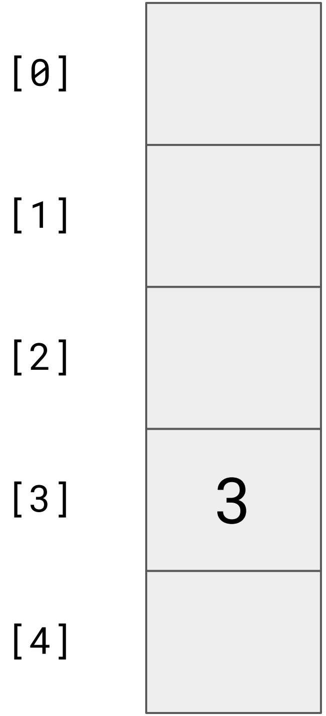


Hash Function:

```
int hash2(int elem) {
    return abs(elem) % numBuckets;
}
```

Input: 3

Hash Code: 3

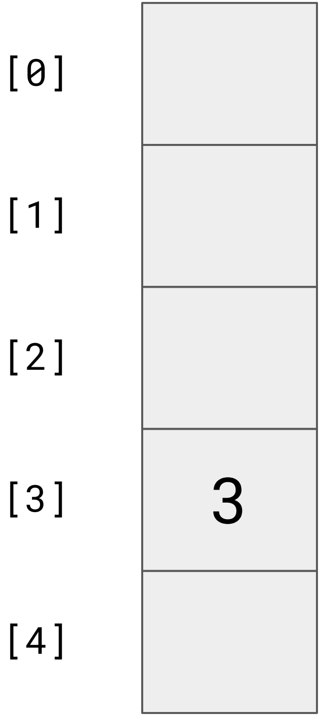


Hash Function:

```
int hash2(int elem) {
    return abs(elem) % numBuckets;
}
```

Input: 3

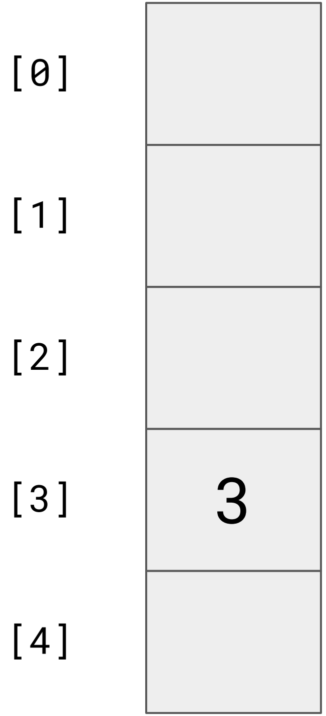
Hash Code: 3



Hash Function:

```
int hash2(int elem) {
    return abs(elem) % numBuckets;
}
```

Input: 17000



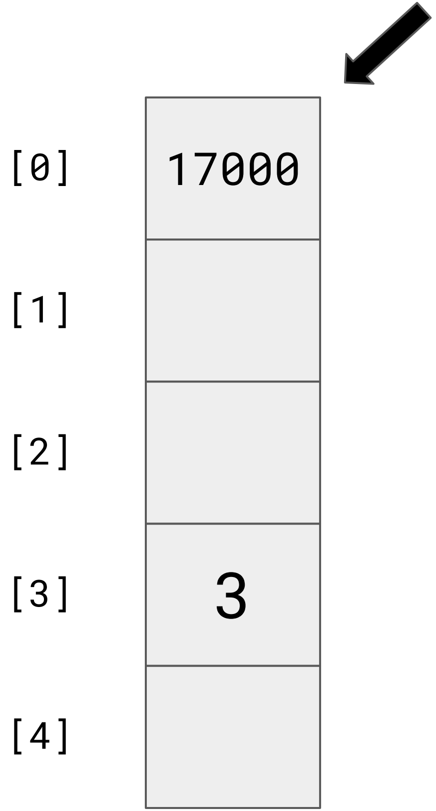
Hash Function:

```
int hash2(int elem) {
    return abs(elem) % numBuckets;
}
```

Input: 17000

Hash Code: 0

Handles this large value!

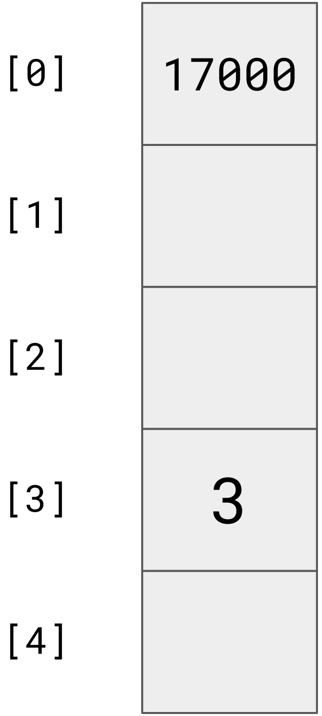


Hash Function:

```
int hash2(int elem) {
    return abs(elem) % numBuckets;
}
```

Input: 17000

Hash Code: 0



Hash Function:

```
int hash2(int elem) {
    return abs(elem) % numBuckets;
}
```

Input: -6

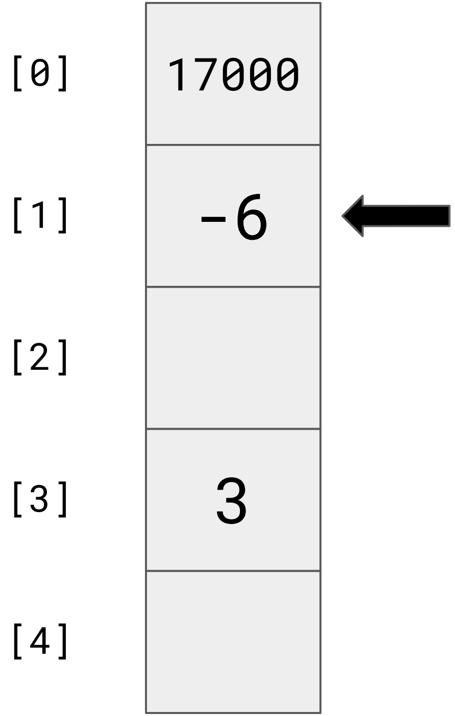
[0]	17000
[1]	
[2]	
[3]	3
[4]	

Hash Function:

```
int hash2(int elem) {
    return abs(elem) % numBuckets;
}
```

Input: -6

Hash Code: 1



Handles this negative value!

Hash Function:

```
int hash2(int elem) {
    return abs(elem) % numBuckets;
}
```

Input: -6

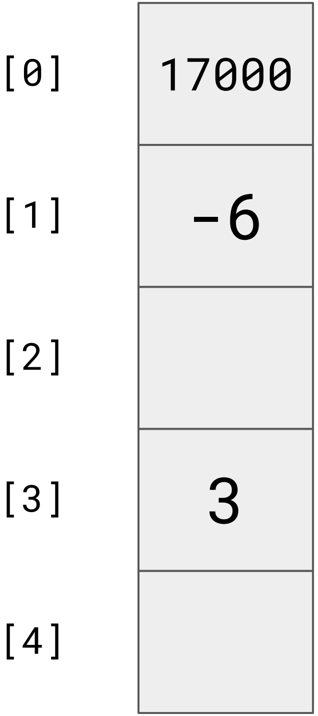
Hash Code: 1

[0]	17000
[1]	-6
[2]	
[3]	3
[4]	

Hash Function:

```
int hash2(int elem) {
    return abs(elem) % numBuckets;
}
```

Input: 8

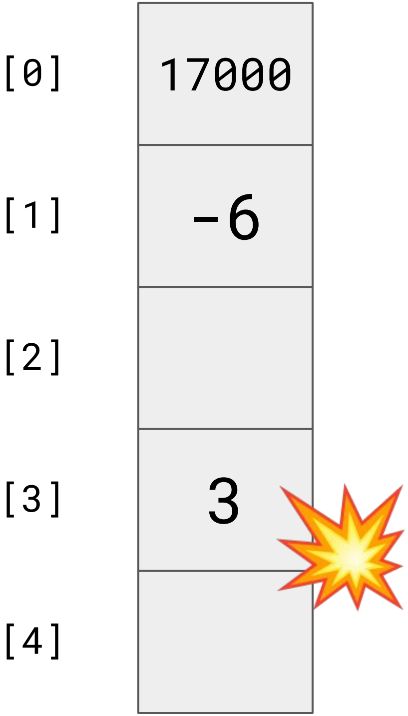


Hash Function:

```
int hash2(int elem) {
    return abs(elem) % numBuckets;
}
```

Input: 8

Hash Code: 3



Hash Function:

```
int hash2(int elem) {
    return abs(elem) % numBuckets;
}
```

Input: 8

Hash Code: 3

Hash Collisions

- Our hash function assigned two different elements to the same bucket

- We call this a collision

Collision Resolution

- We have to decide what to do when collisions happen

- Instead of having our array store `int`, let's have it store `ListNode*` 🤯
  - Each bucket will now be a linked list
  - When we have a collision, we can add the new element to the front of the list in `O(1)`

```
HashTable::HashTable() {
    // Initialize array of buckets
    _elements = new ListNode*[NUM_BUCKETS]();
}
```

A double pointer (ListNode**)! This means that each array element is a pointer. More in CS107!

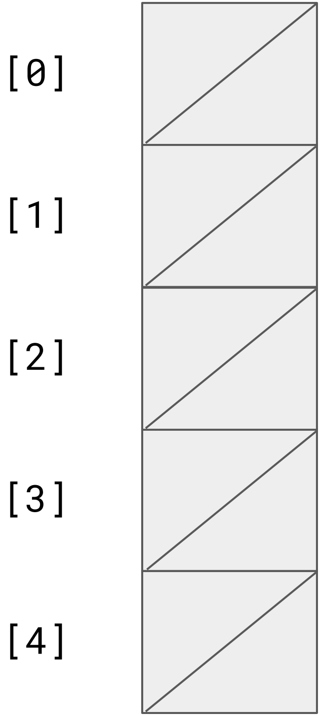
```
HashTable::HashTable() {
    // Initialize array of buckets
    ListNode **_elements = new ListNode*[NUM_BUCKETS]();
}
```

A double pointer! This means that each array element is a pointer. More in CS107!

Initialize each bucket to the nullptr

This is called a

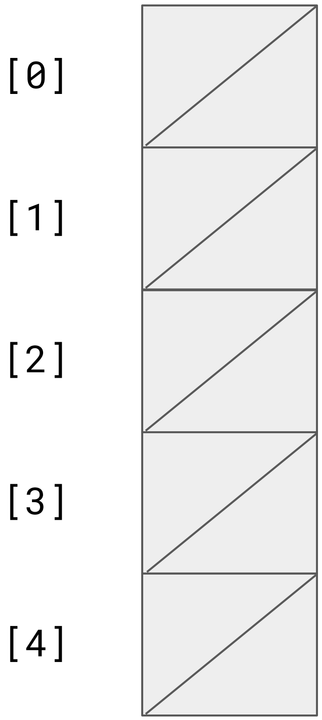
Chaining Hash Table



Hash Function:

```
int hash2(int elem) {
    return abs(elem) % numBuckets;
}
```

Input: 2

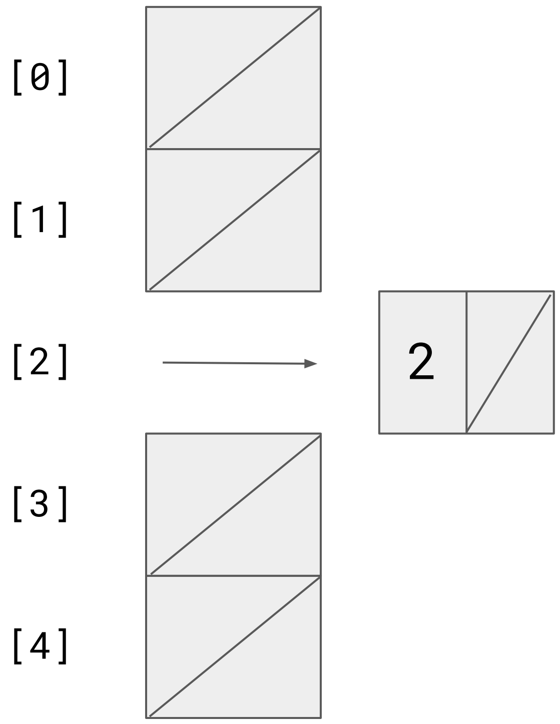


Hash Function:

```
int hash2(int elem) {
    return abs(elem) % numBuckets;
}
```

Input: 2

Hash Code: 2

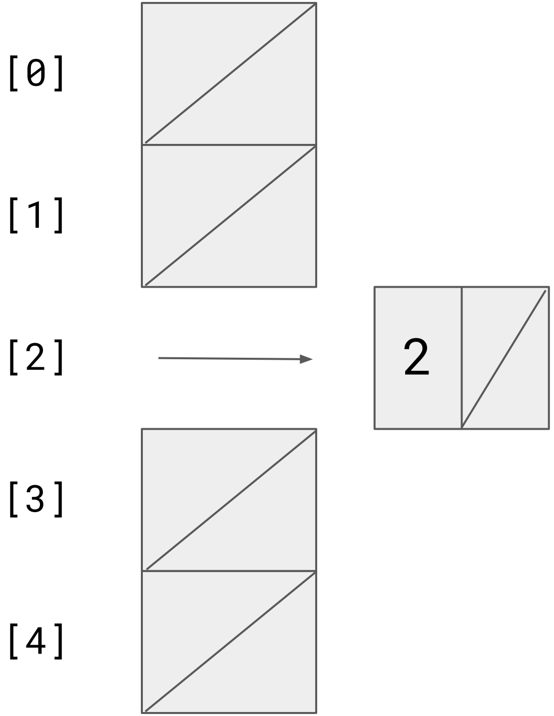


Hash Function:

```
int hash2(int elem) {
    return abs(elem) % numBuckets;
}
```

Input: 2

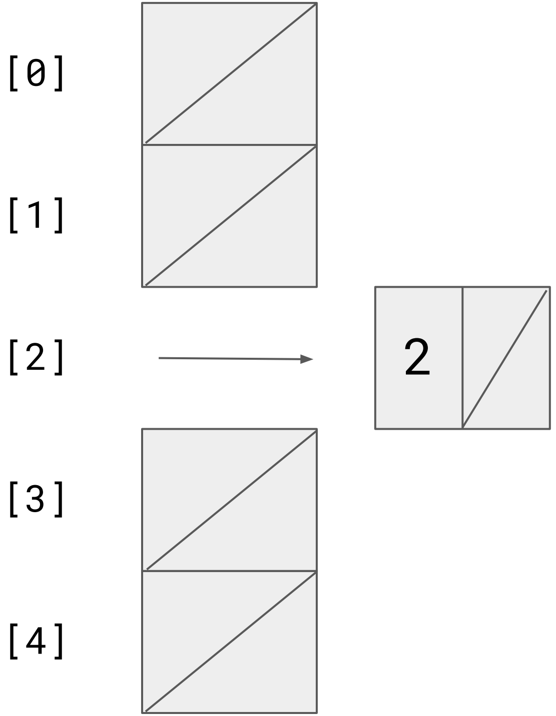
Hash Code: 2



Hash Function:

```
int hash2(int elem) {
    return abs(elem) % numBuckets;
}
```

Input: 10

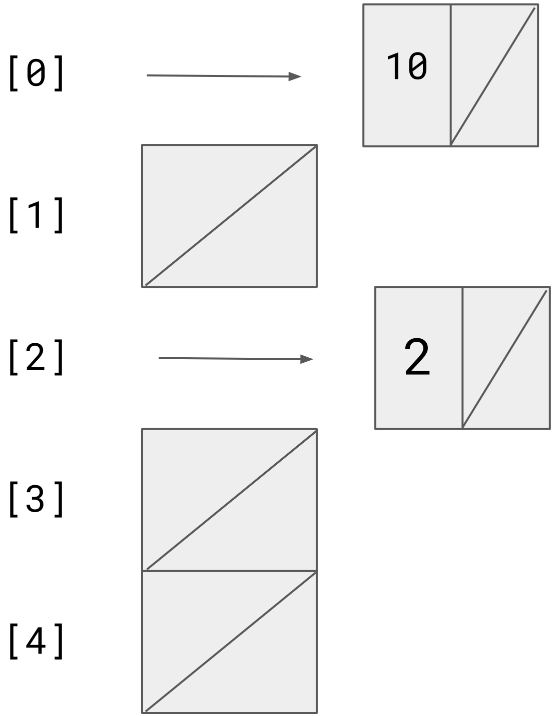


Hash Function:

```
int hash2(int elem) {
    return abs(elem) % numBuckets;
}
```

Input: 10

Hash Code: 0

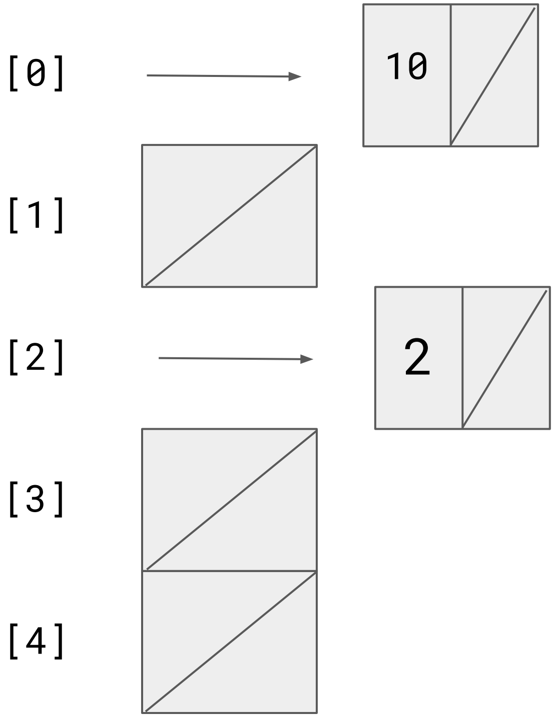


Hash Function:

```
int hash2(int elem) {
    return abs(elem) % numBuckets;
}
```

Input: 10

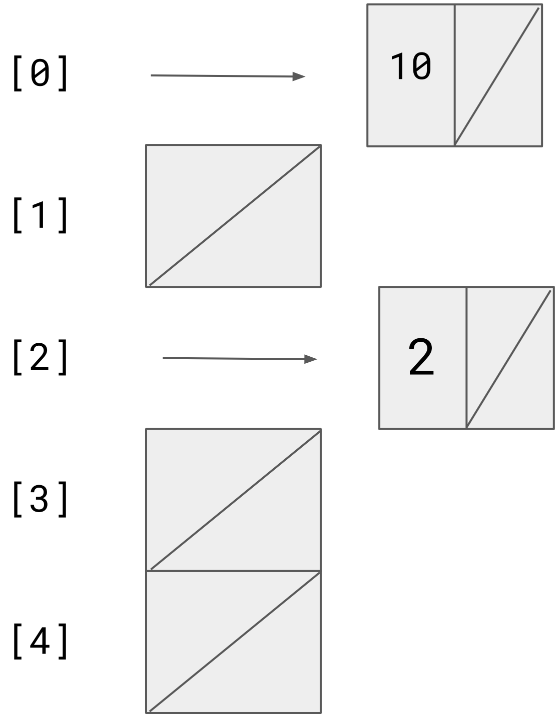
Hash Code: 0



Hash Function:

```
int hash2(int elem) {
    return abs(elem) % numBuckets;
}
```

Input: 7

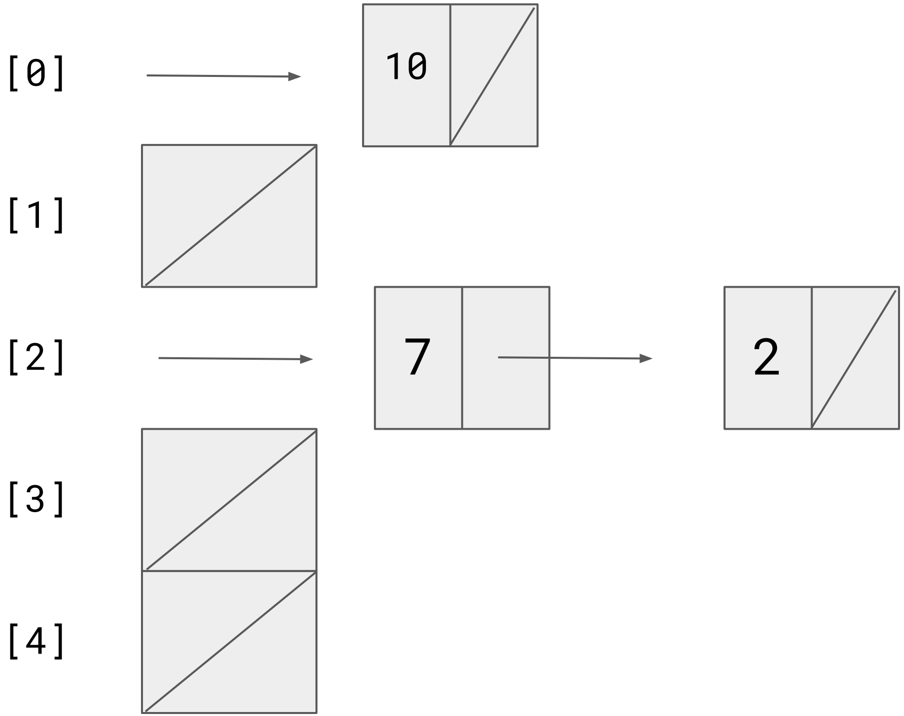


Hash Function:

```
int hash2(int elem) {
    return abs(elem) % numBuckets;
}
```

Input: 7

Hash Code: 2



Hash Function:

```
int hash2(int elem) {
    return abs(elem) % numBuckets;
}
```

Input: 7

Hash Code: 2

Inserting into this chaining hash table is

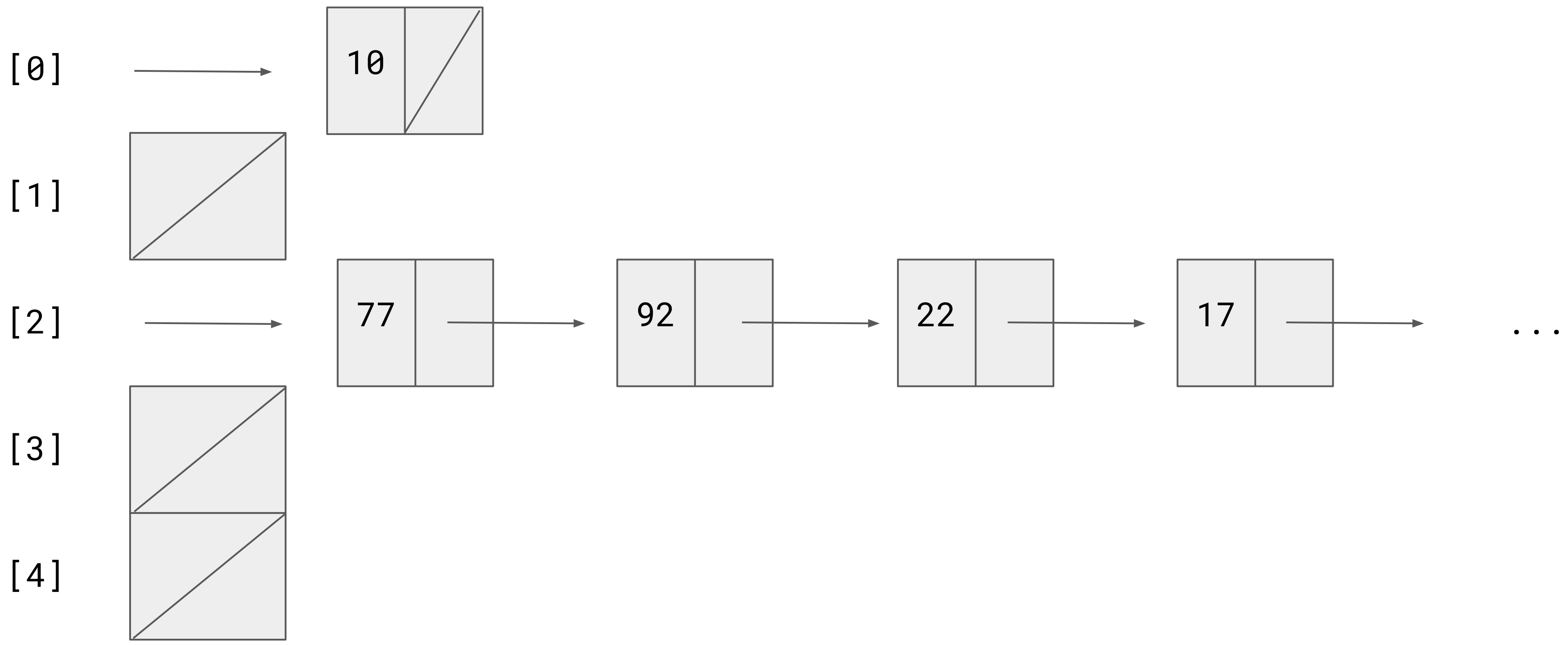
$$O(1)$$

```cpp
void HashTable::insert(int elem) {
    if (contains(elem)) return;
    int bucket = hash2(elem);
    ListNode *front = _buckets[bucket];

    // Create new front of list, tack previous onto end
    ListNode *cur = new ListNode{elem, front};
    _elements[bucket] = cur;
}
```

Say you got the following elements as inputs next:

17, 22, 92, 77


[0]
10
[1]
[2]
77
92
22
17
...
[3]
[4]

With several collisions, our `contains` and `remove` will be

$O(n)$

Where n is the number of elements in the relevant bucket

Our goal is to get a strong hash function that:

- Distributes elements evenly (“spread”)
- Maintains a reasonable load factor

Load Factor

- The average number of elements in each bucket
  - If the load factor is low: wasted space
  - If the load factor is high: slow operations

- The load factor of a hash table with n elements and b buckets is:

$$\frac{n}{b}$$

Strong Hash Functions

- There's tons of research in designing strong hash functions

- Beyond the scope of this class
  - CS161, CS166, CS265

HashSet

Assuming we have a strong hash function

Contains

Add

Remove

HashSet

Assuming we have a strong hash function

Contains	$O(n/b)$
Add	
Remove	

HashSet

Assuming we have a strong hash function

Contains	$O(n/b)$
Add	$O(n/b)$
Remove	

HashSet

Assuming we have a strong hash function

Contains	O(n/b)
Add	O(n/b)
Remove	O(n/b)

With b chosen to be close to n, we can approximate $O(1)$ contains, add, and remove

That’s just about as good as we can do! ✅

The Stanford library `HashSet` and `HashMap` are implemented with hash tables!

HashMap

clear()	O(N)
containsKey(*key*)	O(1)
equals(*map*)	O(N)
firstKey()	O(1)
get(*key*)	O(1)
isEmpty()	O(1)
keys()	O(N)
lastKey()	O(1)
mapAll(*fn*)	O(N)
put(*key*, *value*)	O(1)
remove(*key*)	O(1)

HashSet

add(*value*)	O(1)
clear()	O(N)
contains(*value*)	O(1)
difference(*otherSet*)	O(N)
equals(*set*)	O(N)
first()	O(1)
intersect(*otherSet*)	O(N)
isEmpty()	O(1)
isSubsetOf(*otherSet*)	O(N)
isSupersetOf(*otherSet*)	O(N)
last()	O(1)
mapAll(*fn*)	O(N)
remove(*value*)	O(1)

Other uses of hash functions

Hash Functions

- Broadly, hash functions map a value to a unique integer value
- Presents in several CS domains

Hash Functions

- The magic of hash functions:
  - They can take in any value and boil it down to a unique number
  - Images, ADTs, files, etc.

- Thought question: how would you hash a string?
  - Length?
  - ASCII representation?
  - What about an image?

Hash Functions

Goal: different values should produce very different hash codes

User table (bcrypt)

Username	Password
alice	$2b$10$aQNe4MK0HDhrkus8GZGQL.Nj11nsx12VTMTDBkykiL/jRbb.fJuGC
bob	$2b$10$TSbaMNCCq6.xNkDVszwwhO9Fpb.eeW6aUSIFzGkPoQrs5RahskOUO
charlie	$2b$10$.5KcQQNEfnkPBYxeiqS2ZeePXLT5J30HG7zngfesyGuc0js37X41e
dakotah	$2b$10$l8n7ZLsq13ygE0m3cQ8oEuBjPnGcGBUA4zvJhnsKgyDEZdEd2EFXa

CS145: Data Management and Data Systems

1

Big Scale

Roadmap

Hashing

Sorting

Hashing-Sorting solves "all" known data scale problems :=)

+ Boost with a few patterns -- Cache, Parallelize, Pre-fetch

THE BIG IDEA

Note
Works for Relational, noSQL
(e.g. mySQL, postgres, BigQuery, BigTable, MapReduce, Spark)

Cryptographic Hash Functions

- Hash functions used in a security context
- One-way function: can't reverse
- Most popular: SHA-256
- More in CS155, CS 253, CS255

Have a great break! 🥳

END