CS 106X, Lecture 28
Hashing

Zachary Birnholz

reading:

Programming Abstractions in C++, Chapter 15

Inspired by slides created by Nick Troccoli, Marty Stepp, Keith Schwarz, Cynthia Lee, Chris Piech, Brahm Capoor, Anton Apostolatos, and others.

A thought exercise

Collection Performance

Consider implementing a set with each of the following data structures:

Collection	add(elem)	contains(elem)	remove(elem)
Sorted array			

If we store the elements in an array in **sorted** order:

```
set.add(9);
set.add(23);
set.add(8);
set.add(-3);
set.add(49);
set.add(12);
```

0	1	2	3	4	5	6	7	8	9
-3	8	9	12	23	49	/	/	/	/

Collection Performance

Consider implementing a set with each of the following data structures:

Collection	add(elem)	contains(elem)	remove(elem)
Sorted array	O(N)	O(logN)	O(N)
Unsorted array			

If we store elements in the **next available index**, like in a vector:

```
set.add(9);
set.add(23);
set.add(8);
...
```

0	1	2	3	4	5	6	7	8	9
9	23	8	-3	49	12	/	/	/	/

Collection Performance

Consider implementing a set with each of the following data structures:

Collection	add(elem)	contains(elem)	remove(elem)
Sorted array	O(N)	O(logN)	O(N)
Unsorted array	**O(1)**	O(N)	O(N)
????? array	**O(1)**	**O(1)**	**O(1)**

What would this O(1) array look like? **Where would we want to store each element?**

```
set.add(9);
set.add(23);
set.add(8);
...
```

0	1	2	3	4	5	6	7	8	9
?	?	?	?	?	?	?	?	?	?

Plan For Today

- **O(1)?!?!**
- Hashing and hash functions
- Announcements
- HashSet implementation
  - Collision resolution
  - Coding demo
  - Load factor and efficiency
- Hash function properties

- **Learning Goal 1:** understand the hashing process and what makes a valid/good hash function.
- **Learning Goal 2:** understand how hashing is utilized to achieve O(1) performance in a HashSet.

Plan For Today

- O(1)?!?!
- **Hashing and hash functions**
- Announcements
- HashSet implementation
  - Collision resolution
  - Coding demo
  - Load factor and efficiency
- Hash function properties

Where to store each element?

An idea: just store element *n* at index *n*.

```
set.add(1);
set.add(3);
set.add(7);
```

0	1	2	3	4	5	6	7	8	9
/	1	/	3	/	/	/	7	/	/

- Benefits?
  - add, `contains`, and `remove` are all O(1)!
- Drawbacks?
  - What to do with set.add(11) or set.add(-5), for example?
  - Array might be sparse, leading to memory waste.

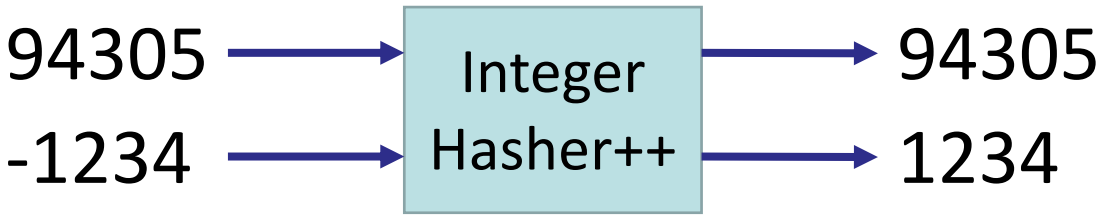
Hashing

- **Hashing:** process of storing each element at a particular predictable index
  - **Hash function:** maps a value to an integer.
    - `int hashCode(Type val);`
  - **Hash code:** the output of a value's hash function.
    - Where the element would go in an infinitely large array.
  - **Hash table:** an array that uses hashing to store elements.

Hash Functions

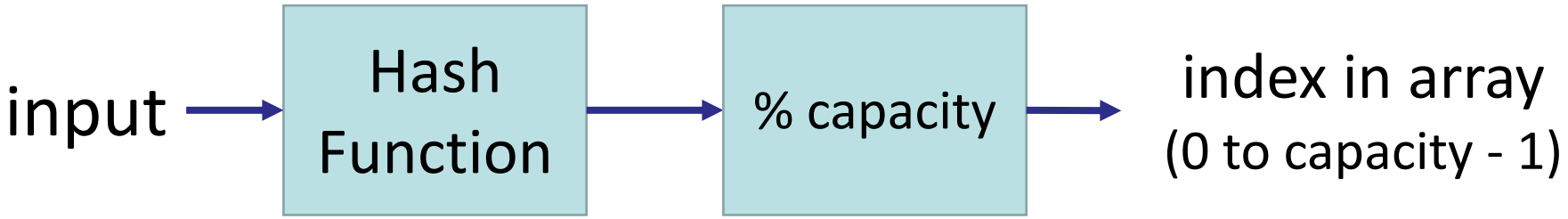
- Our hash function before was hashCode(n) → n.

- To handle negative numbers, hashCode(n) → abs(n):

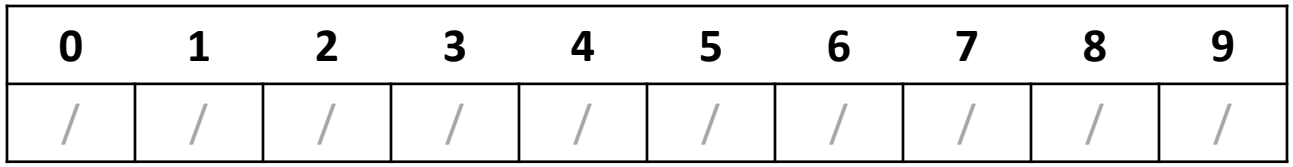


Element placement process

- **To handle large hashes, mod hash code by array capacity**
  - "Wrap the array around"



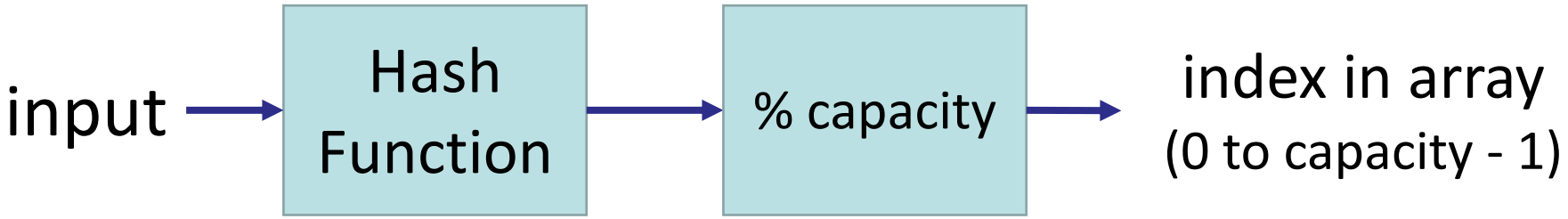

```
set.add(37);
```



Capacity = 10

Element placement process

- **To handle large hashes, mod by array capacity**
  - "Wrap the array around"



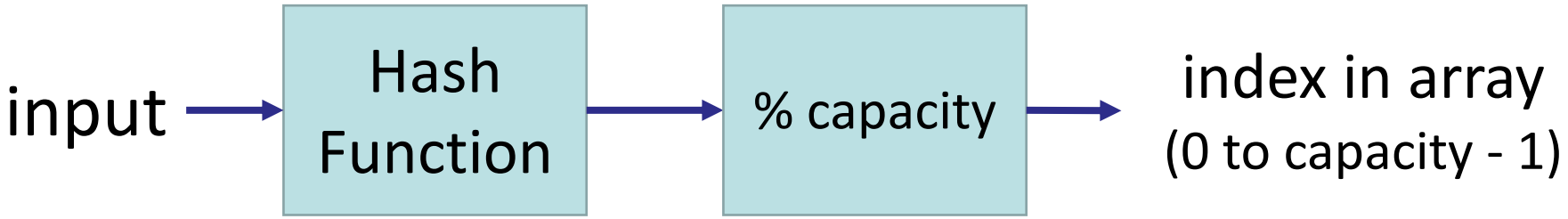

```
set.add(37);        // abs(37) % 10 == 7
```

0	1	2	3	4	5	6	7	8	9
/	/	/	/	/	/	/	37	/	/

Capacity = 10

Element placement process

- **To handle large hashes, mod by array capacity**
  - "Wrap the array around"



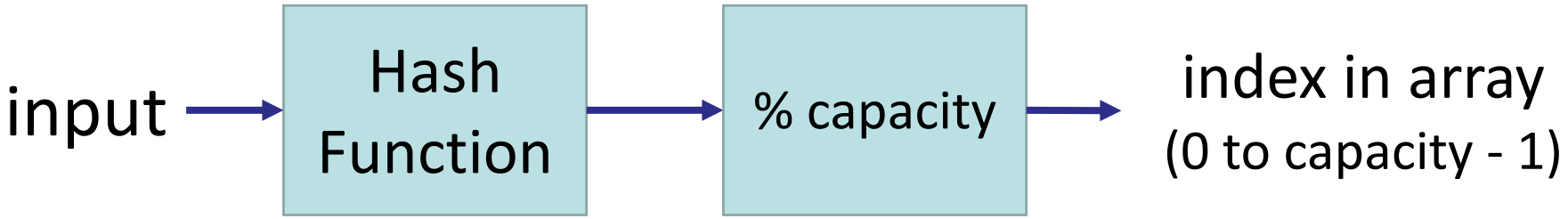

```
set.add(37);        // abs(37) % 10 == 7
set.add(-2);
```

0	1	2	3	4	5	6	7	8	9
/	/	/	/	/	/	/	37	/	/

Capacity = 10

Element placement process

- **To handle large hashes, mod by array capacity**
  - "Wrap the array around"



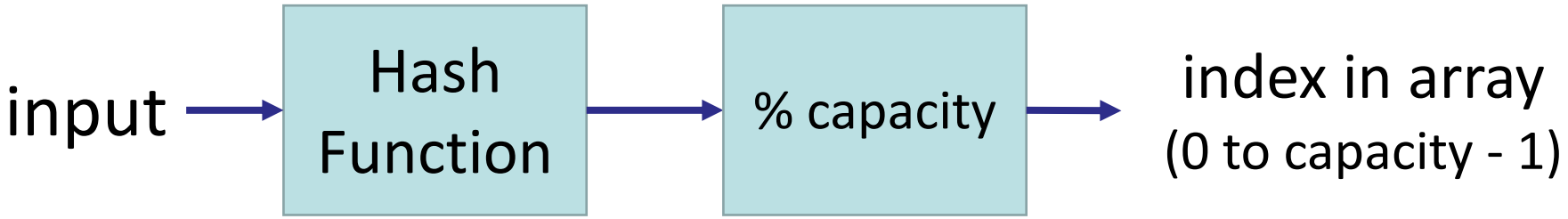

```
set.add(37);        // abs(37) % 10 == 7
set.add(-2);        // abs(-2) % 10 == 2
```

0	1	2	3	4	5	6	7	8	9
/	/	-2	/	/	/	/	37	/	/

Capacity = 10

Element placement process

- **To handle large hashes, mod by array capacity**
  - "Wrap the array around"



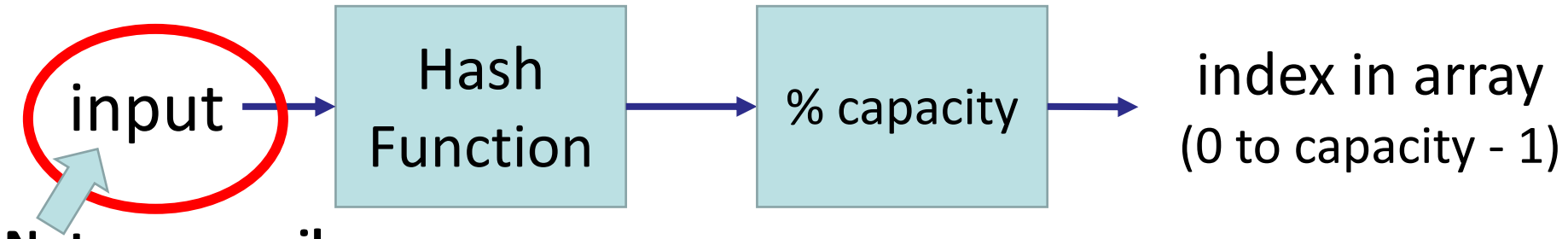

```
set.add(37);        // abs(37) % 10 == 7
set.add(-2);        // abs(-2) % 10 == 2
set.add(49);
```

0	1	2	3	4	5	6	7	8	9
/	/	-2	/	/	/	/	37	/	/

Capacity = 10

Element placement process

- **To handle large hashes, mod by array capacity**
  - "Wrap the array around"



Not necessarily an int!

```
set.add(37);      // abs(37) % 10 == 7
set.add(-2);      // abs(-2) % 10 == 2
set.add(49);      // abs(49) % 10 == 9
```

0	1	2	3	4	5	6	7	8	9
/	/	-2	/	/	/	/	37	/	49

Capacity = 10

Collisions

- **Collision:** when two distinct elements map to the same index in a hash table

```
set.add(37);
set.add(-2);
set.add(49);
set.add(12);        // collides with -2...
```

0	1	2	3	4	5	6	7	8	9
/	/	-2	/	/	/	/	37	/	49

- **Collision resolution:** a method for resolving collisions

Plan For Today

- O(1)?!?!
- Hashing and hash functions
- **Announcements**
- HashSet implementation
  - Collision resolution
  - Coding demo
  - Load factor and efficiency
- Hash function properties

Announcements

- Final exam review session is this **Wed. 12/5 7-8:30pm in Hewlett 103**
- Final exam info page is up on the website, more to come on Wednesday
- Zach's office hours tomorrow are from 2-4pm

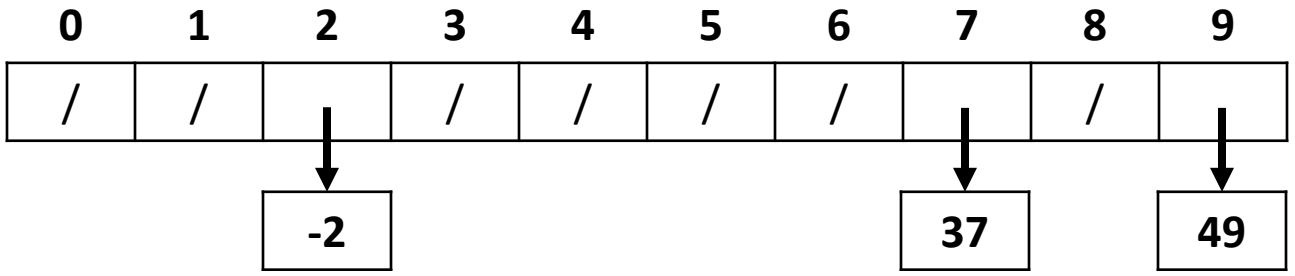
Plan For Today

- O(1)?!?!
- Hashing and hash functions
- Announcements
- **HashSet implementation**
  - **Collision resolution**
  - Coding demo
  - Load factor and efficiency
- Hash function properties

Separate Chaining

- **Separate chaining:** form a linked list at each index so multiple elements can share an index
  - Lists are short if the hash function is well-distributed
  - This is one of many different possible collision resolutions.

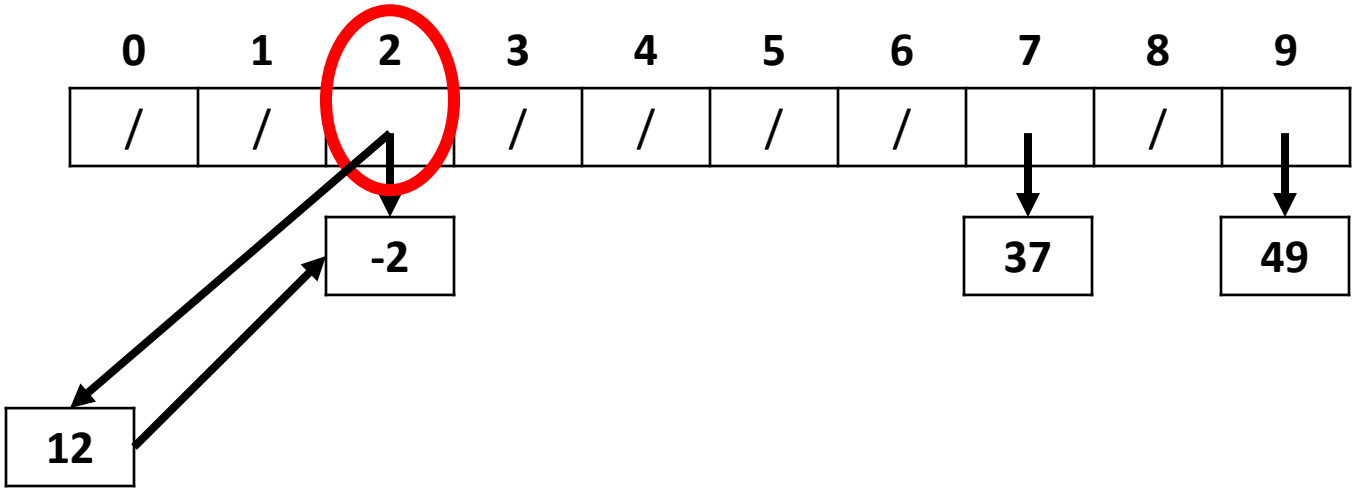
```
set.add(37);
set.add(-2);
set.add(49);
```



Separate Chaining: add

- **Separate chaining:** form a linked list at each index so multiple elements can share an index
  - Just add new elements to the linked lists when adding to HashSet to resolve collisions

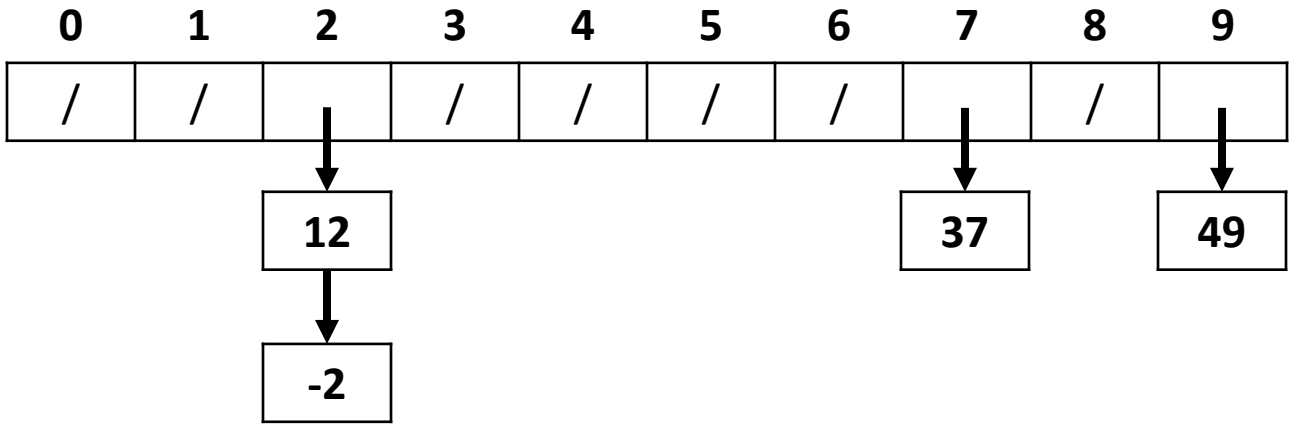
```
set.add(37);
set.add(-2);
set.add(49);
set.add(12);
```



Separate Chaining: add

- **Separate chaining:** form a linked list at each index so multiple elements can share an index
  - Just add new elements to the linked lists when adding to HashSet to resolve collisions

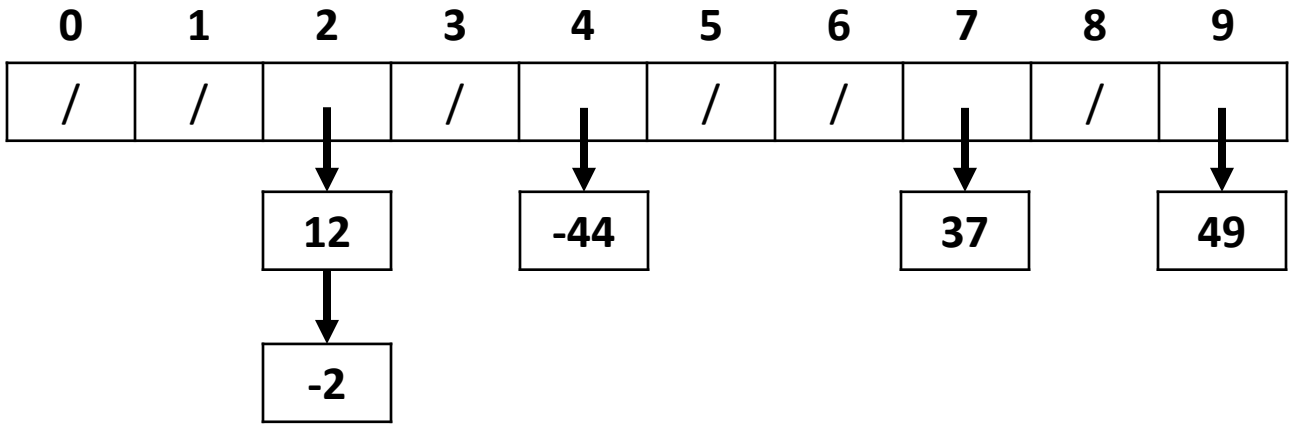
```
set.add(37);
set.add(-2);
set.add(49);
set.add(12);
```



Separate Chaining: add

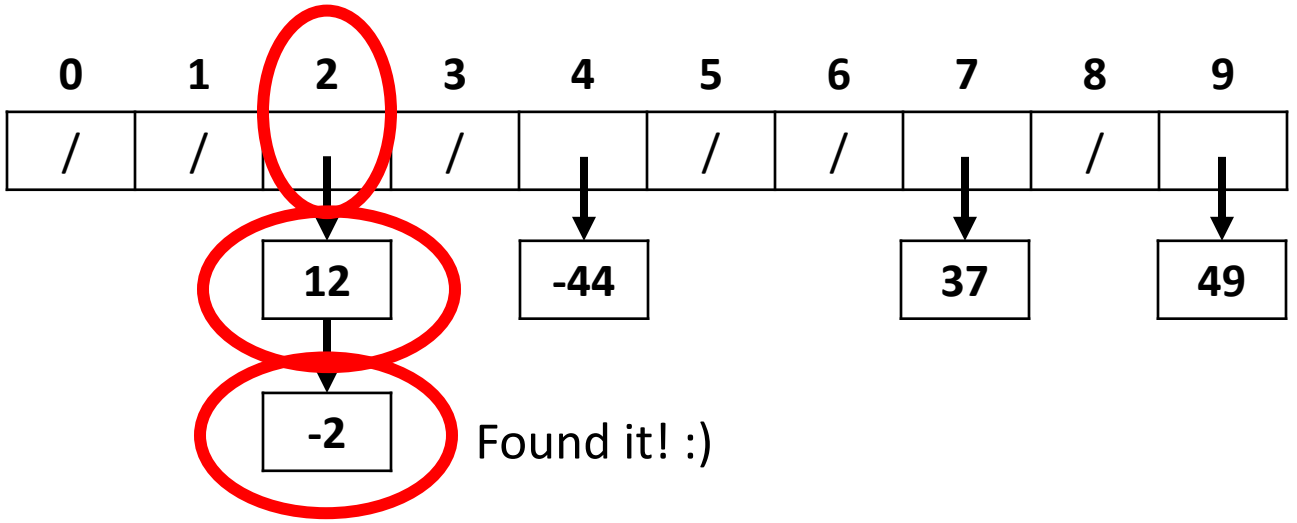
- **Separate chaining:** form a linked list at each index so multiple elements can share an index
  - **For add:** just add new elements to the front of the linked lists when adding to HashSet to resolve collisions

```
set.add(37);
set.add(-2);
set.add(49);
set.add(12);
set.add(-44);
```



Separate Chaining: contains

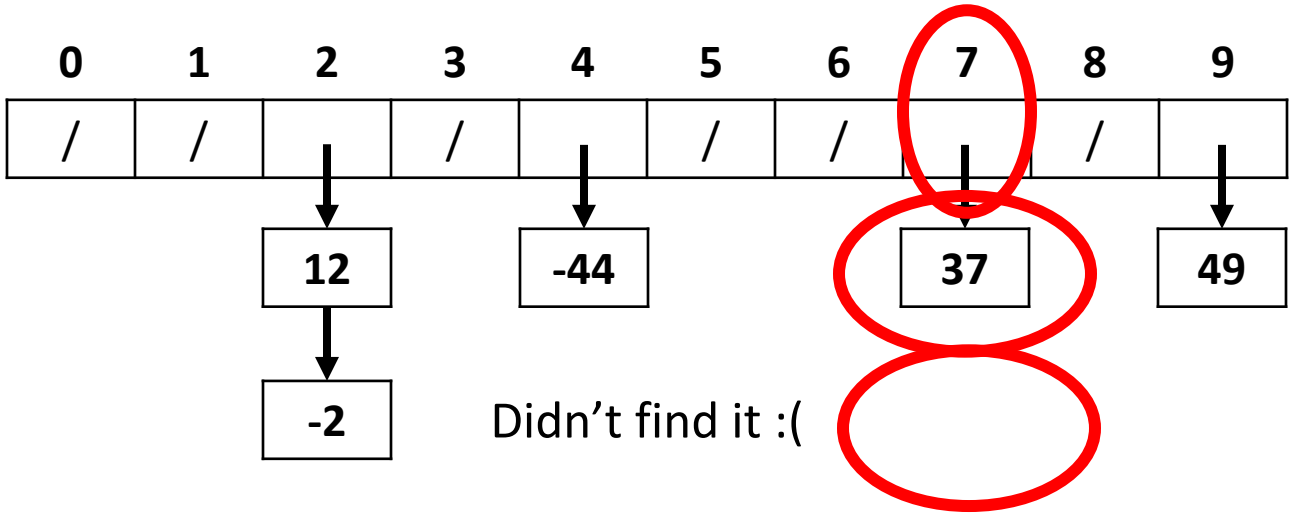
- **Separate chaining:** form a linked list at each index so multiple elements can share an index
  - **For contains:** loop through appropriate linked list and see if you find the element you're looking for




```
set.contains(-2); // true
set.contains(7);  // false
```

Separate Chaining: contains

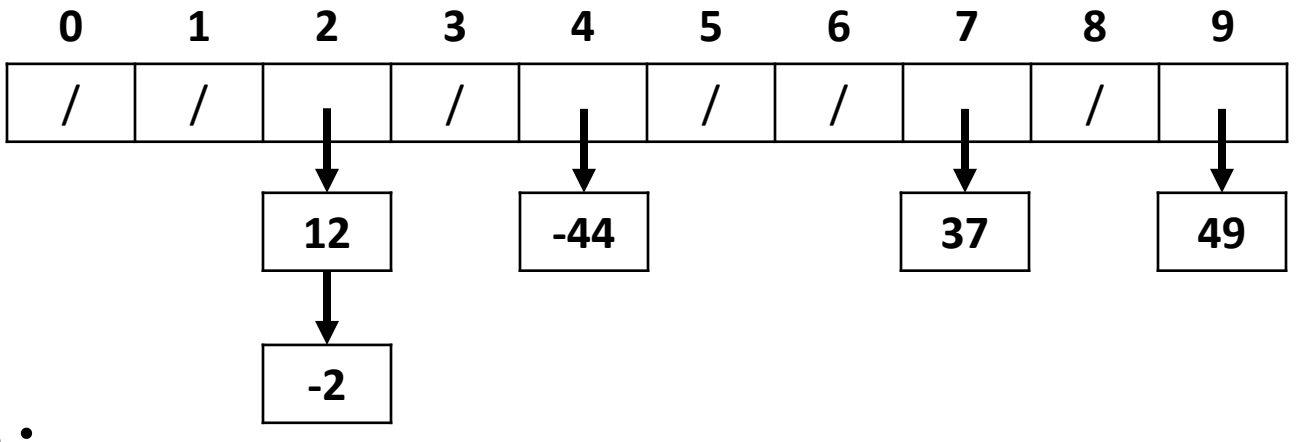
- **Separate chaining:** form a linked list at each index so multiple elements can share an index
  - **For contains:** loop through appropriate linked list and see if you find the element you're looking for




```
set.contains(-2); // true
set.contains(7);  // false
```

Separate Chaining: remove

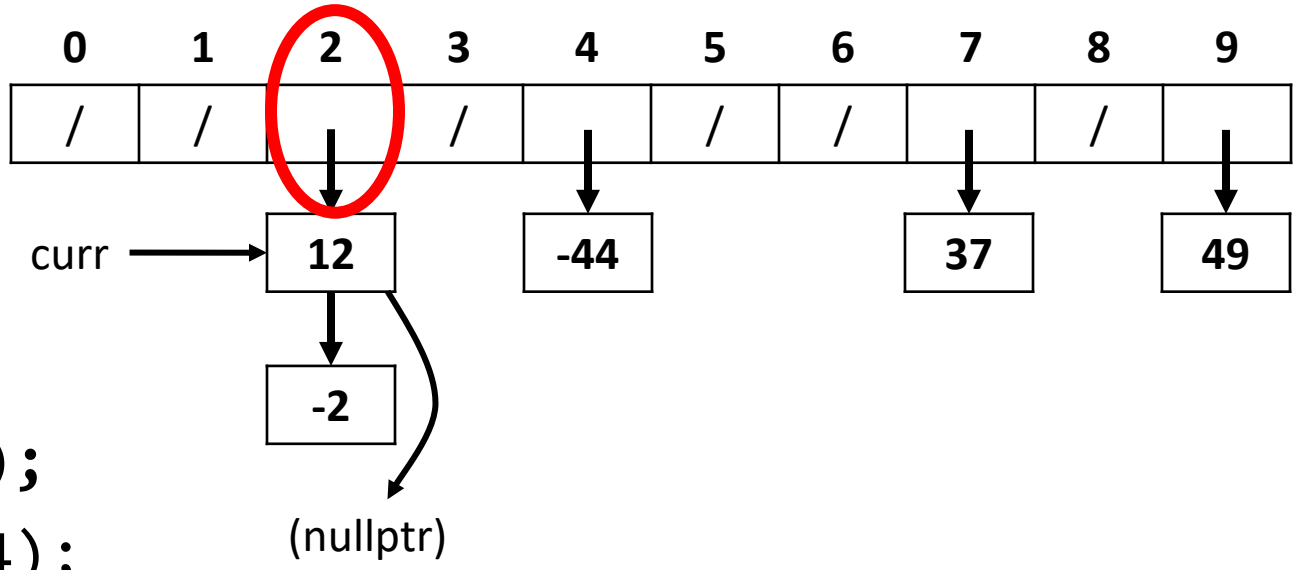
- **Separate chaining:** form a linked list at each index so multiple elements can share an index
  - **For remove:** delete the element from the appropriate linked list if it's there




```
set.remove(-2);
set.remove(-44);
```

Separate Chaining: remove

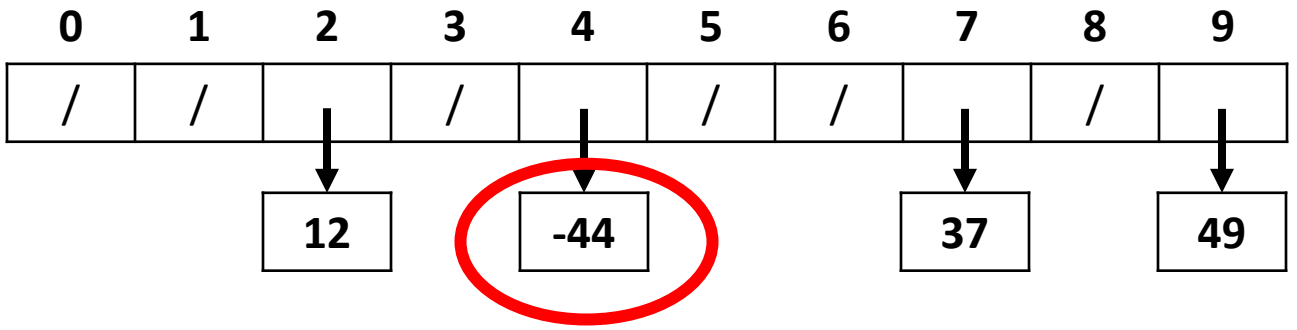
- **Separate chaining:** form a linked list at each index so multiple elements can share an index
  - **For remove:** delete the element from the appropriate linked list if it's there




```
set.remove(-2);
set.remove(-44);
```

Separate Chaining: remove

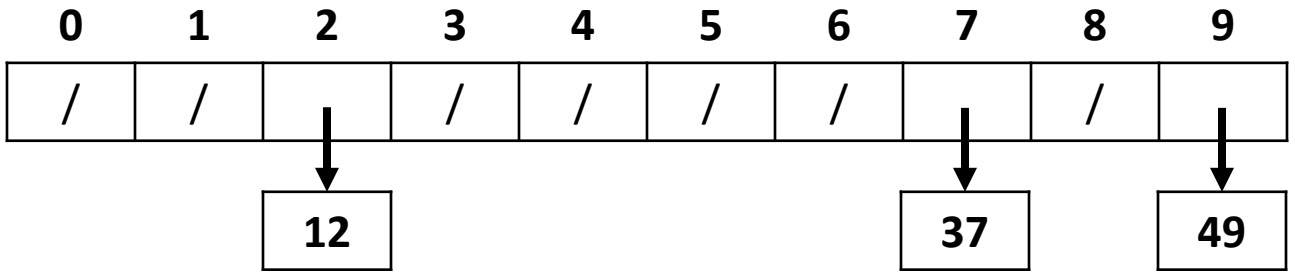
- **Separate chaining:** form a linked list at each index so multiple elements can share an index
  - **For remove:** delete the element from the appropriate linked list if it's there




```
set.remove(-2);
set.remove(-44);
```

Separate Chaining: remove

- **Separate chaining:** form a linked list at each index so multiple elements can share an index
  - **For remove:** delete the element from the appropriate linked list if it's there




```
set.remove(-2);
set.remove(-44);
```

Plan For Today

- O(1)?!?!
- Hashing and hash functions
- Announcements
- **HashSet implementation**
  - Collision resolution
  - **Coding demo**
  - Load factor and efficiency
- Hash function properties

Coding demo

- Let's code a HashSet for integers using separate chaining!

- Note that we will use the following struct in our implementation:

```
struct HashNode {
    int data;
    HashNode* next;
};
```

Solution code 1/3

```
SeparateChainingHashSet::SeparateChainingHashSet(int capacity) {
   this->currSize = 0;
   this->capacity = capacity;
   elems = new HashNode*[capacity]();
}

SeparateChainingHashSet::~SeparateChainingHashSet() {
    clear();
   delete[] elems;
}

void SeparateChainingHashSet::add(int elem) { // note that this doesn't account for re-hashing
   int index = hashCode(elem) % capacity;
    if (!contains(elem)) {
      HashNode* newElem = new HashNode(elem);
      newElem->next = elems[index];
      elems[index] = newElem;
       currSize++;
   }
}
```

Solution code 2/3

```
bool SeparateChainingHashSet::contains(int elem) const {
    int index = hashCode(elem) % capacity;
    HashNode* curr = elems[index];
    while (curr != nullptr) {
        if (curr->data == elem) { return true; }
        curr = curr->next;
    }
    return false;
}
```

Solution code 3/3

```
void SeparateChainingHashSet::remove(int elem) {
   int index = getIndex(elem);
   HashNode* curr = elems[index];
    if (curr != nullptr) {
       if(curr->data == elem) { // elem is at the front of the list
          elems[index] = curr->next;
          delete curr; currSize--;
       } else {
       /* loop through the list in this bucket until we find
        * elem and remove it from the list if it's there */
          while (curr->next != nullptr) {
             if (curr->next->data == elem) {
                HashNode* trash = curr->next;
                curr->next = trash->next;
                delete trash;
                currSize--;
                break;
             }
             curr = curr->next;
          }
       }
    }
}
```

Plan For Today

- O(1)?!?!
- Hashing and hash functions
- Announcements
- **HashSet implementation**
  - Collision resolution
  - Coding demo
  - **Load factor and efficiency**
- Hash function properties

Load Factor

- Question: can a HashSet using separate chaining ever be “full”?
  - It can never be “full”, but it slows down as its linked lists grow

Load Factor

- **Load factor:** the average number of values stored in a single index.

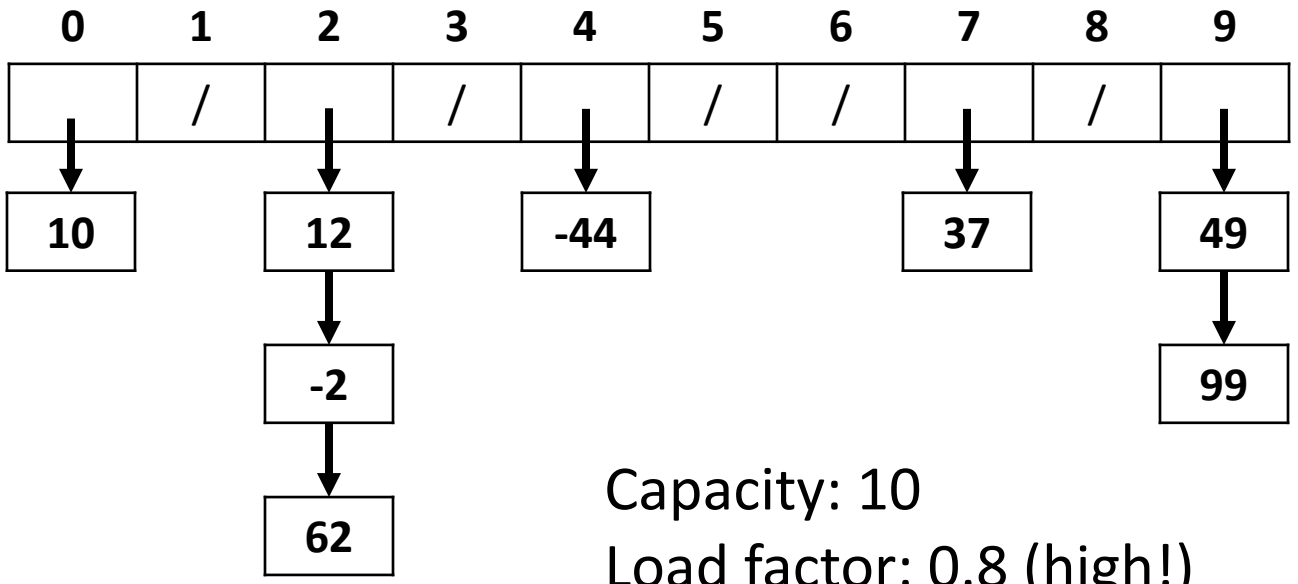
$$load\ factor = \frac{total\ \#\ entries}{total\ \#\ indices}$$

- A lower load factor means better runtime.

- Need to **rehash** after exceeding a certain load factor.
  - Generally after load factor >= 0.75.

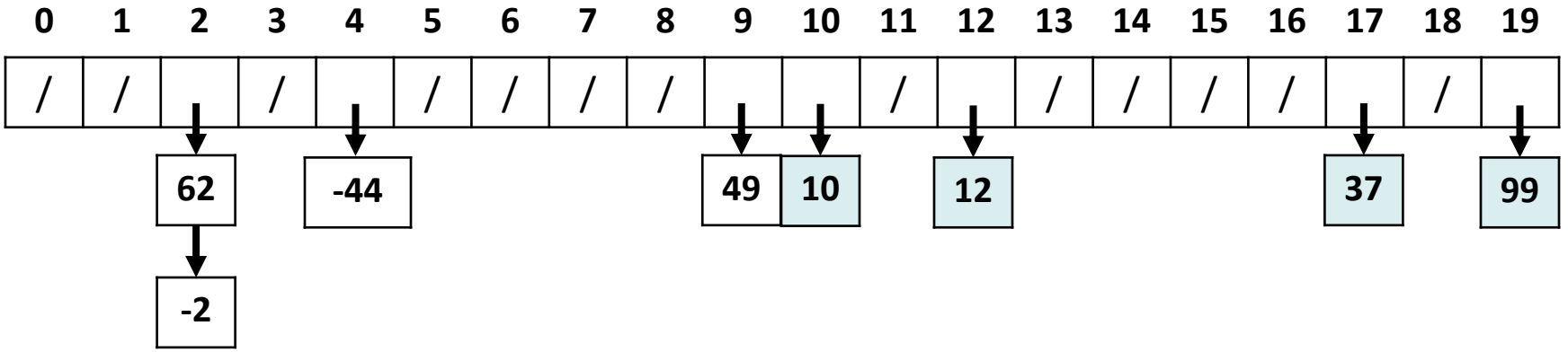
Rehashing

- **Rehashing:** growing the hash table when the load factor gets too high.
  - Can't just copy the old array to the first few indices of a larger one (why not?)



Rehashing

- **Rehashing:** growing the hash table when the load factor gets too high.
  - Loop through lists and re-add elements into new hash table
  - Blue elements are ones that moved indices



Capacity: 20
Load factor: 0.4 (better!)

Plan For Today

- O(1)?!?!
- Hashing and hash functions
- Announcements
- HashSet implementation
  - Collision resolution
  - Coding demo
  - Load factor and efficiency
- **Hash function properties**

Hash function properties

- REQUIRED: a hash function must be **consistent.**
  - Consistent with itself:
    - **hashCode(A) == hashCode(A)** as long as A doesn't change
  - Consistent with equality:
    - **If A == B,** then **hashCode(A) == hashCode(B)**
    - Note that A != B doesn't necessarily mean that hashCode(A) != hashCode(B)

- DESIRABLE: a hash function should be **well-distributed.**
  - A good hash function minimizes collisions by returning mostly unique hash codes for different values.

Hash function properties

- Hash codes can be for any data type (not just for ints)
  - Need to somehow "add up" the object's state.

- A well-distributed hashCode function for a string:

```
int hashCode(string s) {
    int hash = 5381;
    for (int i = 0; i < (int) s.length(); i++) {
        hash = 31 * hash + (int) s[i];
    }
    return hash;
}
```

  - This function is used for hashing strings in Java

Possible hashCode 1

- **Question:** Which of these two hash functions is better?

A.

```
int hashCode(string s) {
    return 42;
}
```

B.

```
int hashCode(string s) {
    return randomInteger(0, 9999999);
}
```

A! Because B is not a valid hash function (B is not consistent).

Possible hashCode 2

- **Question:** Which of these two hash functions is better?

A.

```
int hashCode(string s) {
    return (int) &s;   // address of s
}
```

B.

```
int hashCode(string s) {
    return (int) s.length();
}
```

B! Because A is not valid (A is not consistent, since two equal strings might not be stored at the same memory address).

Possible hashCode 3

- Is the following hash function valid? Is it a good one? Could it have collisions?

```
int hashCode(string s) {
    int hash = 0;
    for (int i = 0; i < (int) s.length(); i++) {
        hash += (int) s[i];   // ASCII value of char
    }
    return hash;
}
```

It's valid, and it's just okay (not as good as Java's, e.g.). This has collisions for strings that are anagrams of each other.

Learning Goals

- **Learning Goal 1:** understand the hashing process and what makes a valid/good hash function.

- **Learning Goal 2:** understand how hashing is utilized to achieve O(1) performance in a HashSet.

- Take CS166 to learn a lot more about different kinds of hashing if you're interested!