- Use any of the dictionaries we've already learned! This gets us $\mathcal{O}(\lg n)$ behavior for each of the operations.

- **Direct Address Table:**

false	false	false	false	false	false	false	false	false
has[0]	has[1]	has[2]	has[3]	has[4]	has[5]	has[6]	has[7]	has[8]

```
void add(int value)           { this.data[value] = true; }
boolean contains(int value) { return this.data[value]; }
void remove(int value)        { this.data[value] = false; }
```

- **BitSet:** Stores one or more ints and uses the ith bit to represent the number i.

$$(1234)_{10} = (00000000000000000000010011010010)_2 = \{1, 4, 6, 7, 10\}$$

```
void add(int value)           { this.set |= 1 << value; }
boolean contains(int value) { return (this.set >> value) & 1; }
void remove(int value)        { this.set &= ~(1 << value); }
```

- Use any of the dictionaries we've already learned! This gets us $\mathcal{O}(\lg n)$ behavior for each of the operations.

- **Direct Address Table:**

false	false	false	false	false	false	false	false	false
has[0]	has[1]	has[2]	has[3]	has[4]	has[5]	has[6]	has[7]	has[8]

```
void add(int value)          { this.data[value] = true; }
boolean contains(int value) { return this.data[value]; }
void remove(int value)       { this.data[value] = false; }
```

- **BitSet:** Stores one or more `int`s and uses the ith bit to represent the number i.

$$(1234)_{10} = (00000000000000000000010011010010)_2 = \{1, 4, 6, 7, 10\}$$

```
void add(int value)          { this.set |= 1 << value; }
boolean contains(int value) { return (this.set >> value) & 1; }
void remove(int value)       { this.set &= ~(1 << value); }
```

Neat Fact: `BitSet`s are often good enough in practice!

Here's some ideas for hash functions for Strings:

- $h(s_0 s_1 \cdots s_{m-1}) = 1$
- $h(s_0 s_1 \cdots s_{m-1}) = \sum_{i=0}^{m-1} s_i$
- $h(s_0 s_1 \cdots s_{m-1}) = 2^{s_0} 3^{s_1} 5^{s_2} 7^{s_3} 11^{s_4} \ldots$
- $h(s_0 s_1 \cdots s_{m-1}) = \sum_{i=0}^{m-1} 31^i s_i$

	Good Dist?	Low compute cost
$h = 1$	No, ok!	O(1) ✓
$\sum s_i$	Yes: what No: where	O(m)
$2^{s_0} 3^{s_1} \ldots$		O(prime(m))
$\sum 31^i s_i$	what! where!	O(m)

Definition (Collision)

A **collision** is when two distinct keys map to the same location in the hash table.

A good hash function attempts to avoid as many collisions as possible, but they are inevitable.

How do we deal with collisions?

There are multiple strategies:

- Separate Chaining
- Open Addressing
  - Linear Probing
  - Quadratic Probing
  - Double Hashing

today

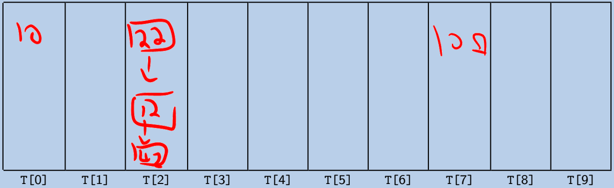
Today, we'll discuss **Separate Chaining**; ~~next time~~, we'll discuss open addressing.

Separate Chaining

Idea

If we hash multiple items to the same location, store a `LinkedList` of them.

Example (Insert: $10, 22, 107, 12, 42$)



Load Factor

Definition (Load Factor (λ))

The **load factor** of a hash table is a measure of "how full" it is. We define it as follows:

$$\lambda = \frac{N}{|T|}$$

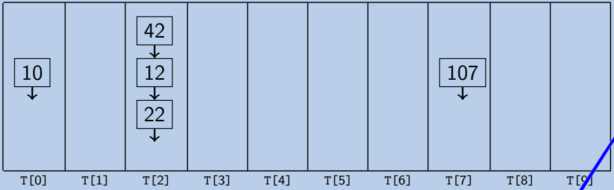
If we're using separate chaining, the average number of elements per bucket is λ.

If we do inserts followed by random finds...

- Each unsuccessful `find` compares against λ items
- Each successful `find` compares against λ items

For separate chaining, we should keep $\lambda \approx 1$

Example (What is the Load Factor?)



What is λ for this hash table?

$n = 5$

$|T| = 10$

$\lambda = 0.5$