Programming Abstractions

CS106B

Cynthia Bailey

Chris Gregg

Today's Topics

Previously (last week):

- Map interface implementation:
  - › Binary Search Tree (BST)

Today: Hashing

- **<u>Also a Map</u>** implementation
- Has pros and cons relative to BST

- **For important announcements, be sure to see the weekly announcements post on the Ed Q&A board! https://edstem.org**
- **Also on Ed: live lecture Q&A with Chris & Jonathan**

Stanford University

Hashing

Implementing the **Map interface (or Stanford `HashMap` class)** with Hashing/Hash Tables

PART 1: Intuition behind the invention of the hash table

Imagine you want to look up your neighbors' names, based on their house number

Home addresses: 10565 Maple St. through 90600 Maple St.

- (roughly 1000 houses—there are varying gaps in house numbers between houses)
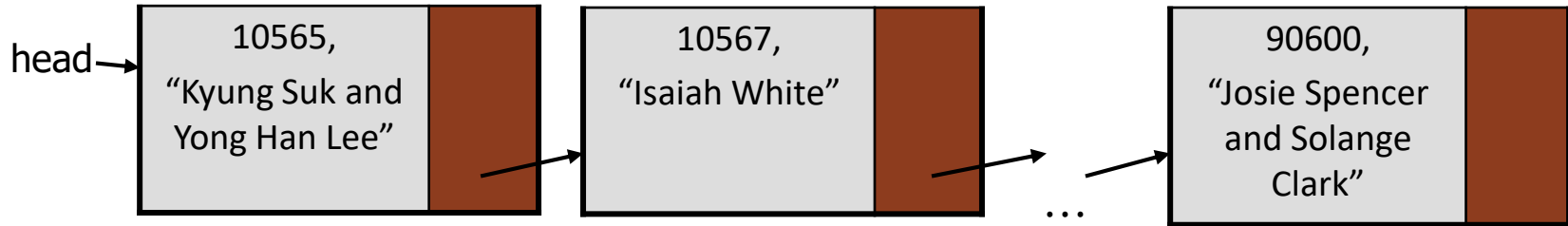- All the houses are on the same street, so we only need to lookup by house number

Names: string containing the name(s) living there

We will consider two data structure options: linked list, and array of strings

Stanford University

Option #1: Linked list

- **Linked list**:
  - Struct has 3 fields: next pointer, int house number, and string name(s)
  - Sort them by house number? (compared sorted/unsorted)

  - Add/remove: O(1) or O(n) depending on sorted/unsorted

  - Find: O(n)

Option #2: Array of strings

- **Array of strings**:
  - `string* addressBook = new string[90601];`

  - Index is house number, string is name
    - The first part of the array will be empty since addresses start at 10565
    - Empty string for any number that is not currently a valid address

Array index (house number)	String value (name)
0	""
1	""
…	…
10565	"Yong Han and Kyung Suk Lee"
10566	""
10567	"Isaiah White"
…	…
90598	""
90599	""
90600	"Josie Spencer and Solange Clark"

Array of Strings

- **Array of strings**:
  - **Add/remove:** ____
    - Ex.: if somebody moves into the vacant house at 90598, how long would it take to update?
  - **Find:** ____
    - Ex.: you want to find the name of the resident at 12475, if any

A. O(1), O(1)
B. O(logn), O(logn)
C. O(n), O(n)
D. Other/none/combination

Array of strings:

Array index *(house number)*	String value *(name)*
0	""
1	""
…	…
10565	"Yong Han and Kyung Suk Lee"
10566	""
10567	"Isaiah White"
…	…
90598	""
90599	""
90600	"Josie Spencer and Solange Clark"

Array of Strings

Array of strings:

- Wow, excellent performance on both!!

- Everything is awesome (?)

- **Discuss:** Can you identify 1-2 specific areas of waste in this approach?
  - Bonus: can you think of a simple fix for at least one of the areas of waste?

Array index *(house number)*	String value *(name)*
0	""
1	""
…	…
10565	"Yong Han and Kyung Suk Lee"
10566	""
10567	"Isaiah White"
…	…
90598	""
90599	""
90600	"Josie Spencer and Solange Clark"

One quick fix:

Array of strings:

```
/* When accessing the array, use array[hash(houseNum)]
 * rather than array[houseNum]. The function hash is
 * just a way to adjust houseNum for efficiency. */
int hash(int houseNumber){
  return houseNumber - 10565;
}
```

- This solves the problem of the enormous gap from 0 to 10565
  - So our array size could be ~80,000 entries instead of 90,600
  - Doesn't solve the problem of gaps between houses
    - How could we do that? A tricky problem…
  - Also, this approach only works for keys of type int

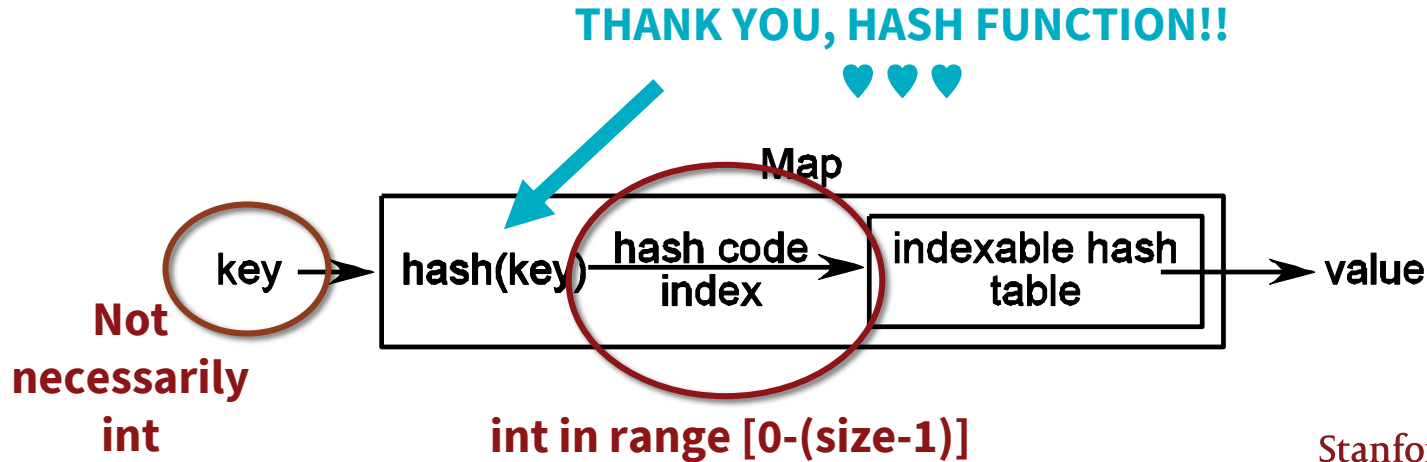| Array index (house number)	String value (name)
0	""
1	""
…	…
10565	"Yong Han and Kyung Suk Lee"
10566	""
10567	"Isaiah White"
…	…
90598	""
90599	""
90600	"Josie Spencer and Solange Clark"

Hashing

Implementing the **Map interface (Stanford `HashMap` class)** with Hashing/Hash Tables

PART 2: Getting the MAGICAL performance of our simple house numbers example on *any* key type, *and* with less waste

Hash Table is just a modified, more flexible array

- **Keys don't have to be integers in the range [0-(size-1)]**
  - › *They don't even have to be integers at all!*
- (Ideally) avoids big gaps like we had with house numbers array
- **Replicates the MAGICAL performance of our array of strings on ANY key/value!!**

THANK YOU, HASH FUNCTION!!
♥ ♥ ♥

Map

key → hash(key) | hash code / index → indexable hash table → value

Not necessarily int

int in range [0-(size-1)]

hash() function

- This is where the **MAGIC** happens!
  - › These are typically mathematically sophisticated functions
  - › They do their best to ensure a nice uniform distribution of elements across the available array (hash table)
  - › They use tricks like modulus (remainder) and prime numbers to do this
  - › A lot of art & science, beyond the scope of this class
  - › Fun times!

Hashing

Implementing the **Map interface (Stanford HashMap class)** with Hashing/Hash Tables

Stanford University

Hash table inserts

Let's pretend we have a profoundly **not**-mathematically-sophisticated hash function:

```
int hash(string key) {
        return key.length();
}
```
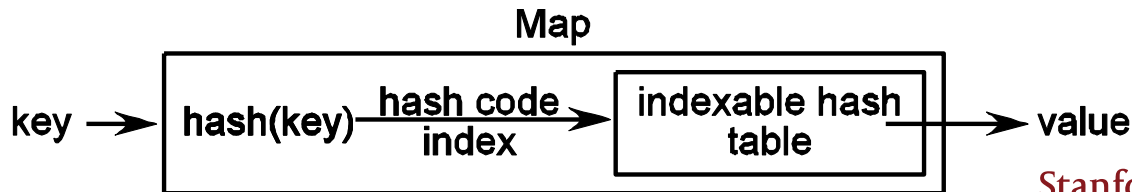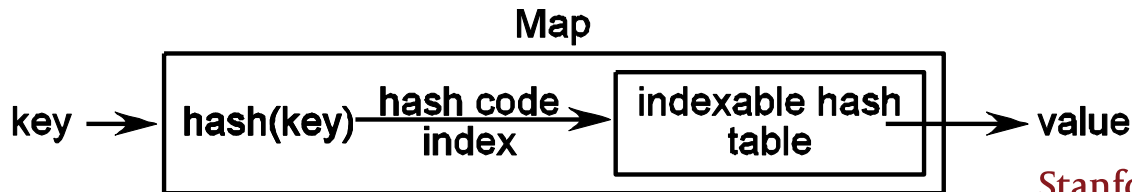
○ Where does key="Annie" value=3 go?

HashMap<string,int> mymap;

mymap["Annie"] = 3;

     See choices in table at right →

     or (E) Some other place

Array index	Hashed data
0	
1	
2	(A) "Annie", 3
3	(B) "Annie", 3
4	(C) "Annie", 3
5	(D) "Annie", 3
6	
7	
8	

Map

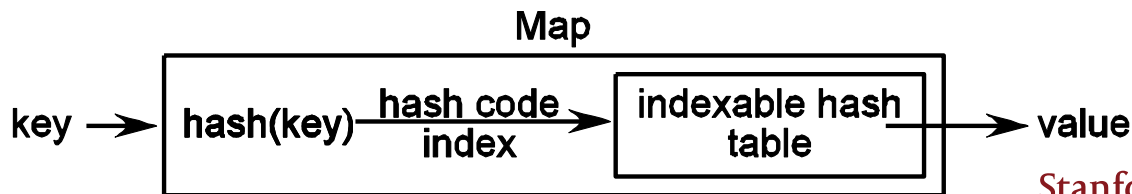key → hash(key) → hash code index → indexable hash table → value

Hash table inserts

Let's pretend we have a profoundly ***not***-mathematically-sophisticated hash function:

```
int hash(string key) {
        return key.length();
}
```

○ Where does key=`"Michael"`, value=`5` go?

`mymap["Michael"] = 5;`

Array index	Hashed data
0	
1	
2	
3	
4	
5	`"Annie", 3`
6	
7	
8	

Map

key → hash(key) —hash code index→ indexable hash table → value

Hash table inserts

Let's pretend we have a profoundly *not*-mathematically-sophisticated hash function:

```
int hash(string key) {
        return key.length();
}
```

- Where does key=`"Michael"`, value=`5` go?

  `mymap["Michael"] = 5;`

Array index	Hashed data
0	
1	
2	
3	
4	
5	`"Annie", 3`
6	
7	`"Michael", 5`
8	

Map

key → hash(key) — hash code / index → indexable hash table → value

Stanford University

Hash table inserts

Let's pretend we have a profoundly ***not***-mathematically-sophisticated hash function:
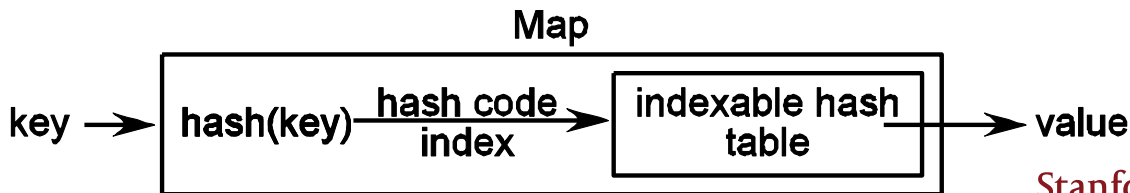
```
int hash(string key) {
    return key.length();
}
```

○ Now insert key="Annie", value=7

```
mymap["Annie"] = 7;
```

See choices in table at right →

or (C) Index 5 should store **both**

"Annie",3 **and** "Annie",7

Array index	Hashed data
0	
1	
2	
3	
4	
5	(A) "Annie", ~~3~~ 7
6	
7	(B) ~~"Michael", 5~~ "Annie", 7
8	

Map

key → hash(key) —hash code index→ indexable hash table → value

Hash table inserts

Let's pretend we have a profoundly ***not***-mathematically-sophisticated hash function:

```
int hash(string key) {
    return key.length();
}
```

- Now insert key=`"Maria"`, value=`8`
  `mymap["Maria"] = 8;`
  See choices in table at right →
  or (D) Index 5 should store **both**
          `"Annie"`,`7` **and** `"Maria"`,`8`

Array index	Hashed data
0	
1	
2	
3	
4	
5	(A) `"Annie", 7` `"Maria", 8`
6	(B) `"Maria", 8`
7	`"Michael", 5`
8	(C) `"Maria", 8`

Map

key → hash(key) — hash code index → indexable hash table → value
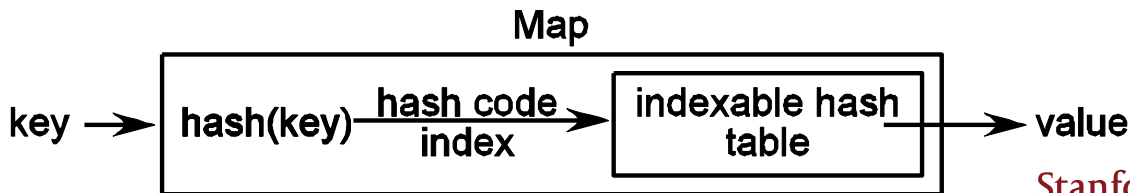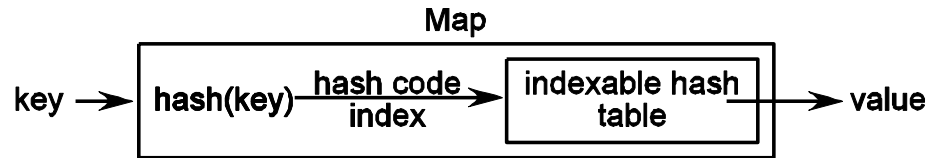
Uh-oh! Hash collisions

We can <u>NOT</u> overwrite the value the way we would if it really were the same key

Can you imagine how you would feel if you used Stanford library HashMap like this and it printed 8?!

```
mymap["Annie"] = 3;
mymap["Annie"] = 7;
cout << mymap["Annie"] << endl; //expect 7, not 3
mymap["Maria"] = 8;
cout << mymap["Annie"] << endl; //expect 7, not 8!
```

Uh-oh! Hash collisions

We may need to worry about *hash collisions*

Hash collision:

- Two keys a, b, a≠b, have the same hash code index (i.e. hash(a) == hash(b))

Need a way of storing multiple values in a given "place" in the hash table, so all user's data is preserved

Uh-oh! Hash collisions

There are two main strategies for resolving this:

1. Put the item in the next bin (as in the (B) choice from our previous slide)—this is called "open addressing"

2. Make each bin be the head of a linked list, and elements can chain off each other as long as needed—this is called "closed addressing" (it will store both, as in the (D) choice from our previous slide)
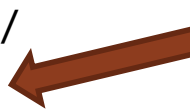
Array index	Hashed data
0	
1	
2	
3	
4	
5	(A) ~~"Annie", 7~~ "Maria", 8
6	(B) "Maria", 8
7	"Michael", 5
8	(C) "Maria", 8

Map

key → hash(key) —hash code index→ indexable hash table → value
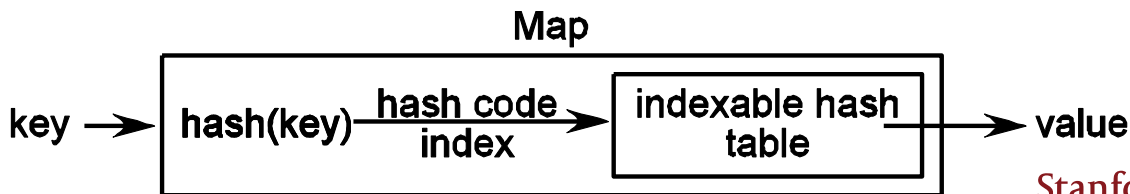
Map Interface: hash-map.h

```
…
private:
  struct Cell {
        KeyType key;
        ValueType value;
        Cell* next;
    };

    /* Instance variables */
    Vector<Cell*> buckets;
    int nBuckets;
    int numEntries;
    int hash(const Key& key) const;
};
```

HashMap

```
…
private:
    struct Cell {
        KeyType key;
        ValueType value;
        Cell* next;
    };

    /* Instance variables */
    Vector<Cell*> buckets;
    int nBuckets;
    int numEntries;
    int hash(const Key& key) const;
};

// Q: Can you draw the HashMap
// object in this memory diagram,
// including filling in values for
// all fields?
```
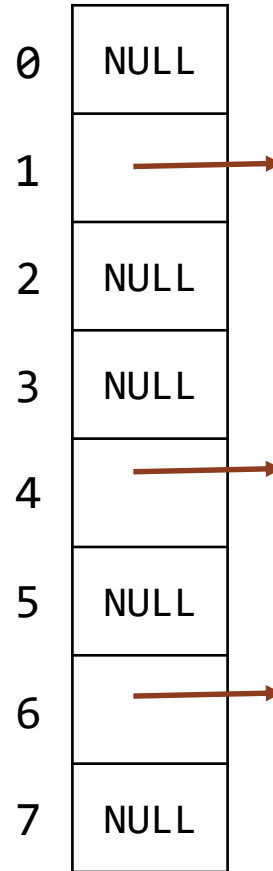
0	NULL
1	→
2	NULL
3	NULL
4	→
5	NULL
6	→
7	NULL

Hash key collisions & Big-O of HashMap

If there are no collisions, find/add/remove are all O(1)—just compute the key and go!

Two factors for ruining this magical land of instantaneous lookup:

- Too-small table (worst case = 1)

- Hash function doesn't produce a good spread

```
int awfulHashFunction(string input) { // h/t http://xkcd.com/221/
    return 4;
  }
```

- Find/add/remove all O(n) worst case

Stanford University