Programming Abstractions

CS106X

Cynthia Lee

Topics Overview

Recently:

- Priority Queue implementations:
  - linked list
  - heap
- Map interface implementation:
  - Binary Search Tree (BST)
- While we're doing trees, another kind of tree:
  - Huffman coding trees (used for compression of files)

Today: Hashing:

- Another Map interface implementation
  - Background: Lookup tables
  - Details: Hashing

Hashing

Implementing the **Map interface** 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

House numbers: 10565 through 90600

- (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

Option #1: Linked list

- **Linked list**:
  - Struct has 3 fields: next pointer, int data (house number), and string data (name)
  - Sort them by house number
  - Add/remove: O(n)
  - Find: O(n)

Option #2: Array of strings

- **Array of strings**:
  - Index is house number, string is name
  - Add/remove:
  - Find:

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**:
  - Index is house number, string is name(s)

Array of strings:

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**:
  - Index is house number, string is name(s)
  - **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:

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

- **Wow, excellent performance on both!!**
- **Only way to do better than O(1) is a time machine that can go back in time and make it take zero/negative time!**
- **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 of strings:

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:

```
/* When accessing the array, use array[hash(houseNum)]
 * rather than array[houseNum]
 */
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…
  - This approach only works for keys of type int

Array of strings:

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"

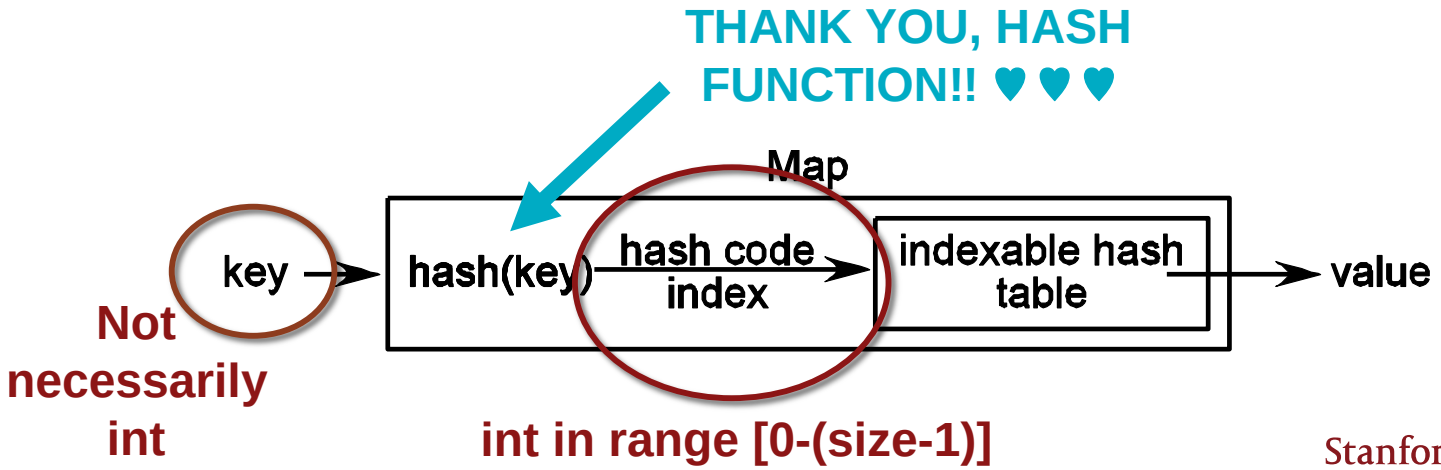
Hashing

Implementing the **Map interface** 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!!**



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** with Hashing/Hash Tables

Hash table inserts

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

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

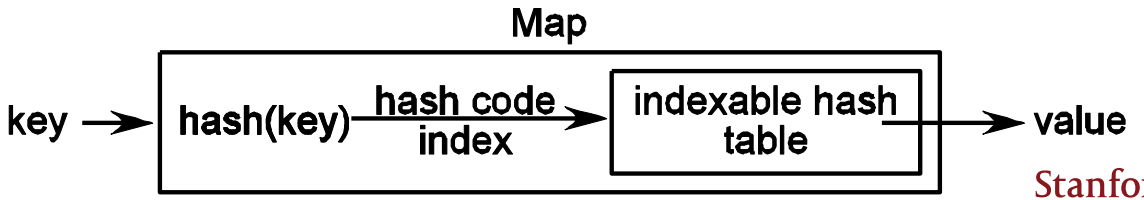
- Where does `key="Annie" value=3` go?

```
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	



Hash table inserts

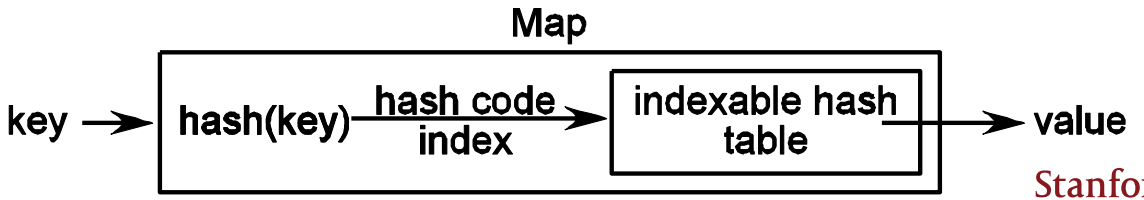
Let's pretend we have a profoundly ***not***-mathematically-sophisticated hash function:

```
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	



Hash table inserts

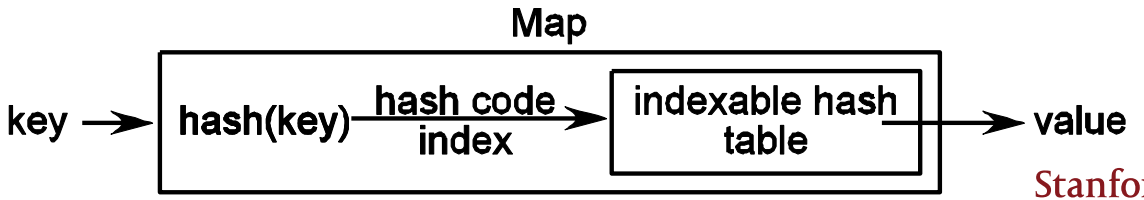
Let's pretend we have a profoundly ***not***-mathematically-sophisticated hash function:

```
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	



Hash table inserts

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

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

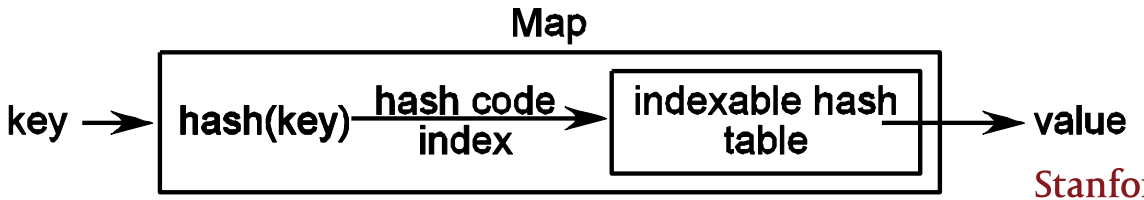
- Now insert key="Annie", value=7

  `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	



Hash table inserts

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

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

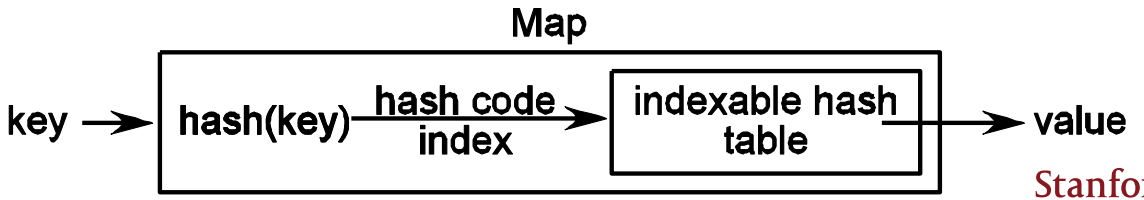
- Now insert key="Maria", value=8

  `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



Uh-oh! Hash collisions

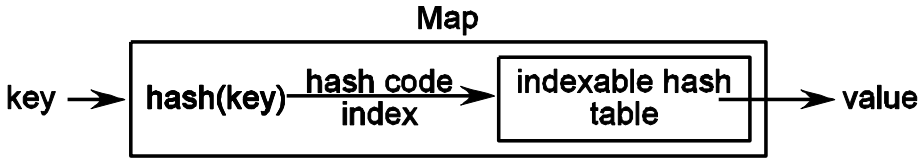
We can NOT overwrite the value the way we would if it really were the same key

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))

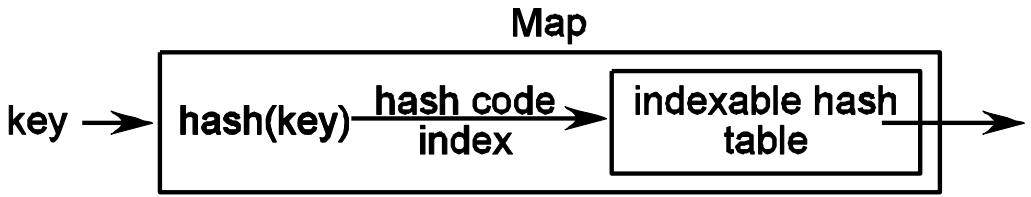
Need a way of storing multiple values in a given “place” in the hash table, so all user’s data is preserved

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”

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 Interface: hash-map.h

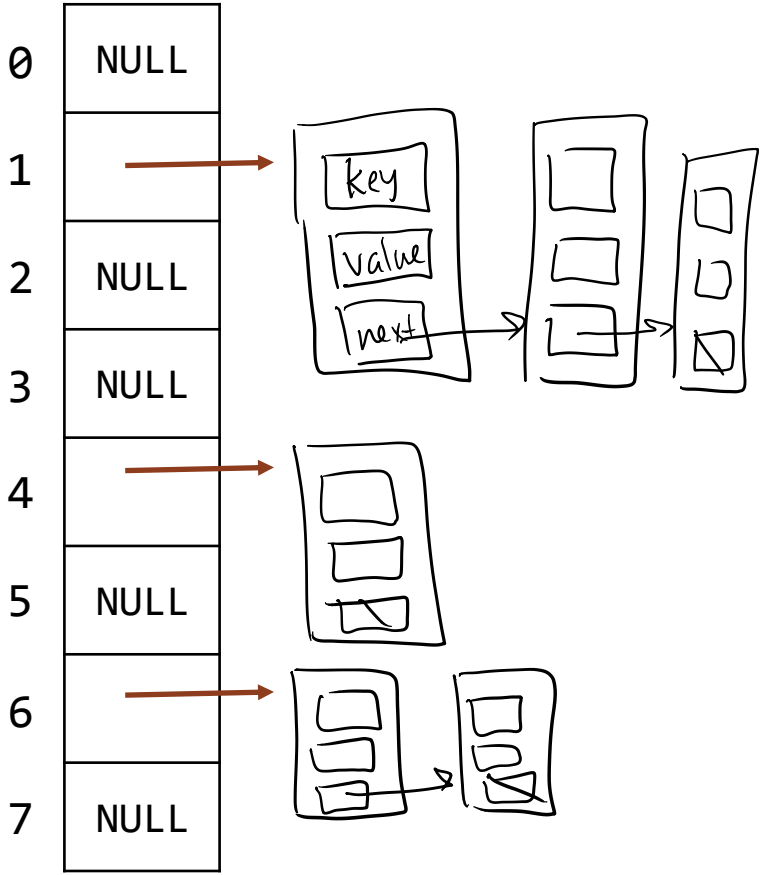
```
…
private:
 struct node {
    Key key;
    Value value;
    node *next;
 };

 node **buckets;
 int numBuckets;
 int count;
 int hash(const Key& key) const;
};
```

HashMap

```
…
private:
 struct node {
    Key key;
    Value value;
    node *next;
 };

 node **buckets;
 int numBuckets;
 int count;
 int hash(const Key& key) const;
};
// Q: Can you draw the HashMap
// object in this memory diagram,
// including filling in values for
// all fields?
```



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) {
      return 4;                          // h/t http://xkcd.com/221/
  }
  ```
- Find/add/remove all O(n) worst case
  ```

The Birthday Problem

Commonly mentioned fact of probability theory:

In a group of n people, what are the odds that 2 people have the same birthday?

- (assume birthdays are uniformly distributed across 365 days, which is wrong in a few ways)

- For n ≥ 23, it is more likely than not
- For n ≥ 57, >99% chance

Moral of the story: hash tables almost certainly have at least one collision