Section 6. Class Design and Hashing

Section materials curated by Neel Kishnani, drawing upon materials from previous quarters.

This week's section exercises explore the ins and outs of content from weeks 6 and 7, focusing on class design and hash functions!

Each week, we will also be releasing a Qt Creator project containing starter code and testing infrastructure for that week's section problems. When a problem name is followed by the name of a **.cpp** file, that means you can practice writing the code for that problem in the named file of the Qt Creator project. Here is the zip of the section starter code:

 Starter code

Problem One: Shrinking the Stack

In class, we implemented the **OurStack** class, which represents a stack of integers. The class definition is shown here:

```
class OurStack {
public:
    OurStack();
    ~OurStack();

    void push(int value);
    int pop();
    int peek() const;

    int size() const;
    bool isEmpty() const;

private:
  void grow();

  int* elems;
  int logicalSize;
  int allocatedSize;
};
```

The implementation is on the next page, just as a refresher. Here's the .cpp file:

```
#include "OurStack.h"
#include "error.h"
const int kInitialSize = 4;

OurStack::OurStack() {
  elems = new int[kInitialSize];
  logicalSize = 0;
  allocatedSize = kInitialSize;
}

OurStack::~OurStack() {
  delete[] elems;
}

void OurStack::push(int value) {
  if (logicalSize == allocatedSize) {
    grow();
  }
  elems[logicalSize] = value;
  logicalSize++;
}

int OurStack::peek() const {
  if (isEmpty()) {
    error("What is the sound of one hand clapping?");
  }
  return elems[logicalSize - 1];
}

int OurStack::pop() {
  int result = peek();
  logicalSize--; return result;
}

int OurStack::size() const {
  return logicalSize;
}

bool OurStack::isEmpty() const {
  return size() == 0;
}

void OurStack::grow() {
  int* newArr = new int[2 * allocatedSize];
  for (int i = 0; i < size(); i++) {
        newArr[i] = elems[i];
    }
  delete[] elems;
  elems = newArr;
  allocatedSize *= 2;
}
```

This problem consists of some questions about the OurStack type:

i. What is the meaning of the term “logical size?” How does it compare to the term “allocated size?” What’s the relationship between the two?

Solution

---

ii. What is the `OurStack()` function? Why is each line in that function necessary?

Solution

---

iii*. What is the `~OurStack()` function? Why is each line in that function necessary? Why isn’t there any code in there involving the logicalSize or allocatedSize data members?

Solution

---

iv. In the `OurStack::grow()` function, one of the lines is `delete[] elems`, and in the next line we immediately write `elems = newArr;`. Why is it safe to do this? Doesn’t deleting `elems` make it unusable?

Solution

---

v. The `OurStack::pop()` function doesn’t seem to have any error-checking code in it. What happens if you try to pop off an empty stack?

Solution

---

vi. What is the significance of placing the `OurStack::grow` function in the private section of the class? What does that mean? Why didn’t we mark it public?

Solution

Right now, our stack doubles its allocated length whenever it runs out of space and needs to grow. The problem with this setup is that if we push a huge number of elements into our stack and then pop them all off, we’ll still have a ton of memory used because we never shrink the array when it’s mostly unused.

vii. Explain why it would not be a good idea to cut the array size in half whenever fewer than half the elements are in use.

Solution

---

viii. Update the code for `OurStack` so that whenever the logical length drops below one quarter of the allocated length, the array is reallocated at half its former length. As an edge case, ensure that the allocated length is always at least equal to the initial allocated capacity. This setup maintains the (amortized) $O(1)$ cost of each insertion and deletion and ensures that the memory usage is always $O(n)$.

Solution

Problem Two: Hash Functions

In CS106B, we’ll use hash functions without spending too much time talking about their internal workings. (Building a good hash function is a challenging endeavor!) To give you a sense about why this is, we’d like you to investigate four different possible hash functions.

Each of the functions shown below is designed to take as input a string and then output a hash code in the range from 0

to 9, inclusive. For each proposed hash function, tell us whether it's

1. not even a valid hash function;
2. a valid hash function, but not a very good one; or
3. a very good hash function.

Of course, you should be sure to justify your answers.

```
int hashFunction1(const string& str) {
  return randomInteger(0, 9); // a value between 0 and 9, inclusive
}
int hashFunction2(const string& str) {
  return 0; // Who cares about str, anyway?
}
```

Solution

These next three functions use the fact that each char has an associated integer value in C++. For example, on most systems the character `'A'` has value 65, the character `'3'` has value 51, the character `'!'` has value 33, and the character `'~'` has value 126. We can use these numeric values to design hash functions for strings.

```
int hashFunction3(const string& str) {
  return str[0] / 10;
}
int hashFunction4(const string& str) {
  return str[0] % 10;
}
int hashFunction5(const string& str) {
  int result = 0;
  for (char ch: str) {
    result += ch; // Again, uses the numeric value of ch
  }
  /* Numeric values for characters can be positive or negative. */
  if (result < 0) {
    result = 0;
  }
  if (result >= 10) {
    result = 9;
  }
  return result;
}
```

Solution

Problem Three: Salting and Hashing

In lecture, we talked about how when storing passwords, it's common to not store the passwords themselves, but rather the hash code of the password using some hash function. This ensures that if someone manages to steal the password database, they can't immediately read off all the passwords stored in the database.

This approach – storing the hashes of the passwords rather than the passwords themselves – is much better than just storing the passwords themselves, but it's not what's done in practice because it doesn't work well if the users have weak passwords.

Specifically, suppose that you have a list of the 100,000 most commonly used passwords, which isn't too hard to find with a bit of Googling. You also have, probably nefariously, stolen the password file from a website, which is represented as a **`Map<string, int>`** mapping each user name to the hash of their password. You could then do the following to figure out the passwords for anyone using weak accounts:

1. Compute the hash codes for the 100,000 commonly-used passwords that you know of.
2. Look at the **`Map<string, int>`** mapping website users to the hashes of their passwords. For each hash code that matches one of the hashes you've computed above, output their password.

Write a function

```
Map<string, string> breakWeakPasswords(const Map<string, int>& stolenPasswordFile,
                                       const Set<string>& knownWeakPasswords,
                                       HashFunction<string> hashFn);
```

that uses the above approach to figure out the passwords of each user who has a weak password, returning a **`Map`** associating each such user with their password. The final parameter, **`hashFn`**, represents the hash function used to compute password hashes. Then, answer the following question: if there are *n* users on the website and there are *m* weak passwords, what is the runtime of the code that you wrote, assuming hash codes can be computed in time O(1)?

In practice, it's most common to use a strategy called **salting and hashing**, which works as follows, to prevent people from recovering passwords in the way you just did. We store for each user two pieces of information:

- a randomly-chosen string called the **salt**, and
- the hash code of the string **`salt + password`**, where password is the user's password.

It's important that the salt is chosen randomly for each user, rather than having a fixed salt used for all passwords.

First, explain why the introduction of a per-user salt means that you can't break passwords using the code that you wrote above. Then, answer the following question: suppose you had the master password file for a website, including the salted, hashed passwords, and you wanted to find all users who have weak passwords. If there are *n* users and *m* weak passwords, how long would it take to find all users who have weak passwords? Represent your answer in big-O notation.

(A note on this problem: we're not teaching this to you because we want you to do Nefarious Things like this. Rather, we wanted you to see what Roguish Scalawags could do to compromise a password system and to see why we take the protective steps that we take to stop them from doing so.)

Solution