CS106X
Autumn 2015 Cynthia Lee

Section 4 (Week 5) Handout

Problem and solution authors include Jerry Cain

Problem 1: Superheroes Then and Now

Analyze the following program, starting with the call to **`elektra`**, and draw the state of memory at the two points indicated. Be sure to differentiate between stack and heap memory, note values that have not been initialized, and identify where memory has been orphaned.

```
struct superhero {
   int wonderwoman;
   superhero *isis;
   int *superman[2];
};

static void elektra() {
   superhero marineboy[2];
   superhero *ironman;
   ironman = &marineboy[1];
   marineboy[0].wonderwoman = 152;
   marineboy[0].superman[0] = new int[2];
   marineboy[0].superman[1] = &(ironman->wonderwoman);
   ironman->superman[0] = marineboy[0].superman[1];
   ironman->superman[1] = &(marineboy[0].superman[0][1]);
   *(ironman->superman[1]) = 9189;
   marineboy[1].isis = ironman->isis = ironman;
```

⇐ First, draw the state of memory just prior to the call to `barbarella`.

```
   barbarella(marineboy[1], ironman->isis);
}

static void barbarella(superhero& storm, superhero *& catwoman) {
   storm.wonderwoman = 465;
   catwoman->isis = &storm;
   catwoman->wonderwoman = 830;
   catwoman->isis[0].superman[1] = &(storm.isis->wonderwoman);
   catwoman = &storm;
   catwoman->wonderwoman = 507;
   catwoman->isis = new superhero[2];
```

⇐ Second, draw the state of memory just before `barbarella` returns

```
}
```

Problem 2: Bloom Filters and Sorted String Sets

For this problem, you're going to implement a **SortedStringSet** class that lays on top of a **Set<string>**. The **public** interface and part of the **private** section are presented here:

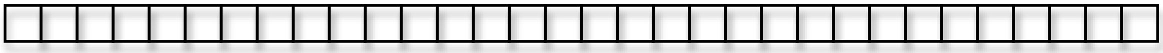
```
class SortedStringSet {
public:
   SortedStringSet(const Vector<int (*)(const std::string&, int)>& hashers);
   ~SortedStringSet();

   int size() const { return values.size(); }

   bool contains(const std::string& value) const;
   void add(const std::string& value);

private:
   Set<std::string> values;
};
```

Beyond its **Set<string>** impersonation, the **SortedStringSet** is engineered so calls to **contains** run very, very quickly for the vast majority of **string**s that **aren't** present. If you know ahead of time that you'll be storing a large number of **string**s, and you expect an overwhelming percentage of **contains** calls to return **false**, you can enhance the **SortedStringSet** so those calls to **contains**—the ones returning **false**—run in time that has absolutely nothing to do with the **Set**'s size.

This enhancement—the one optimizing **contains** to return **false** more quickly—can be realized with a **Bloom filter**.

The Bloom filter is an array-based structure that determines whether a **string** may or can't be present. It's basically a Boolean array, where all values are initially **false**. Each time a **string** is inserted, several hash functions—all provided at construction time—are applied to produce a series of hash codes. The **string** then leaves several footprints—one for each hash code—on the Bloom filter by ensuring the Boolean at each hash code is **true**.

To illustrate, assume the filter is of size 32. An empty box represents **false**, and a filled one represents **true**. Initially, all Booleans are **false**, so all boxes are empty:

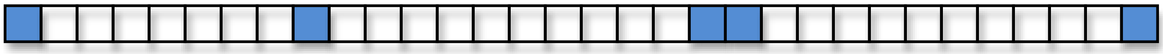


Further assume the **SortedStringSet** constructor was given **five** exceptionally good hash functions. Each time a **string** is inserted, the hash functions are applied to the **string** to produce five hash codes. The corresponding slots in the array are all set to **true**—those are the footprints—before being inserted into the companion **Set**. Each time you search a

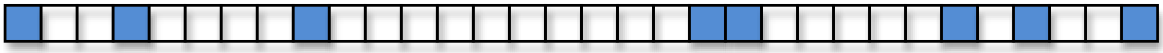
SortedStringSet, you first confirm all relevant footprints are present, lest you waste precious clock cycles searching the traditional **Set**.

Here's a play-by-play illustrating how this works:

- Imagine you've installed five hash functions at construction time, and the first word you insert is **"computer".** Further imagine the five functions hash **"computer"** to 20, 8, 0, 19, and 31. The Bloom filter would evolve into the following:



- Imagine you then insert the word **"tower"**, and it hashes to 3, 28, 3, 19, and 26 (of course, two or more of the hash functions produce the same hash code from time to time, and there's no way to prevent that.) The Bloom filter would evolve into this:



- If later on you search for **"tower"** or **"computer"**, you'd first confirm the expected footprints are accounted for, and once you do that, you'd take the time to search the **Set<string>** to verify the word is actually present.

- When searching for an arbitrary word, you first determine whether or not the word's hash-generated footprints are present. If not, you can quickly return **false** without examining the. If all footprints are present, you would search the **Set** to see if the word is really there.

The full class file for the **StringSortedSet** will maintain a Bloom filter in addition to the **Set<string>**. The **Set<string>** stores the values, but the implementation of **contains** doesn't even look to the **Set** unless the Bloom filter suggests it might be there. The Bloom filter is used to efficiently guard against relatively time-consuming **Set** searches when it can tell searching the **Set** is a waste of time.

The full interface for the **SortedStringSet** is the following:

```
class SortedStringSet {
public:
   SortedStringSet(const Vector<int (*)(const std::string&, int)>& hashers);
   ~SortedStringSet();

   int size() const { return values.size(); }

   bool contains(const std::string& value) const;
   void add(const std::string& value);

private:
   Set<std::string> values;
   // additional fields and methods you'll add to support the Bloom filter
};
```

For the lab, implement the constructor, the destructor, and the **contains** and **add** methods. You must specify what additional **private** data members and methods are needed to support the Bloom filter. While coding, adhere to the following:

- The vector of hash functions passed to the constructor will contain at least one function pointer. The function pointers (of type **int (*)(const std::string&, int)**—that's the notation for it) address functions that takes the **string** being hashed and the size of the Bloom filter.
- The Bloom filter should be implemented as a dynamically allocated array of Booleans, initially of length 1001.
- The implementation of **contains** can return **false** for one of two reasons. The first reason is that the Bloom filter makes it clear a **string** was never inserted, because one of more of its footprints is missing. The second reason is that the **Set** just doesn't contain the **string** even though it survived the filter.
- If when adding a new **string** you notice that the number of **true**s exceeds the number of **false**s, you should allocate a new, larger Boolean array (you choose the size), dispose of the old one, and effectively reinsert all of the **string**s as if the new array was the original. That means every once in a while, a call to **add** is very expensive, but the reallocation happens so infrequently that it doesn't impact the average running time. (We do this, else the Bloom filter will become congested with mostly **true**s, and will become less and less effective at filtering.)

Some thought questions:

- Why do we need to store the actual **string**s in an underlying **Set<string>**? Isn't it enough to just store its footprints as evidence that a string is present?
- Given the above design constraints, why can't we implement a **remove** method all that easily? How could you change the implementation so that **remove** could be supported more easily?