CS107 Handout 25
Spring 2008 May 7th, 2008

CS107 Midterm Exam

This is an open-note exam. You can refer to any course handouts, handwritten lecture notes, and printouts of any code relevant to a CS107 assignment.

Those taking the exam remotely should phone in if they have questions. Once you're done, fax the exam to Stanford, but hold on to the original until you get the graded fax copy back.

Good luck!

`leland`
username: _____

Last Name: _____

First Name: _____

I accept the letter and spirit of the honor code. I've neither given nor received aid on this exam. I pledge to legally change my name to McLovin if my handwriting is deemed illegible.

(signed) _____

		Score	Grader
1. Your Green Grocer	(15)	_____	_____
2. **`hashset`**: Take II	(15)	_____	_____
3. **`decompress`**	(10)	_____	_____
Total	**(40)**	_____	_____

Problem 1: Your Friendly Green Grocer (15 points: 5, 5, and 5)

Consider the following **`struct`** definitions:

```
   typedef struct {
      int apple;
      char *banana;
      char cherry[16];
   } fruit;

   typedef struct {
      short pea[6];
      fruit potato;
      fruit *parsnip[3];
   } veggie;

   fruit *tort(fruit **fig, int grape);
   fruit *casserole(veggie carrot, veggie *spinach)
   {
      fruit date;
line 1   date.cherry[4] = spinach->pea[carrot.potato.apple];
line 2   ((veggie *)(((veggie *)carrot.parsnip[0])->parsnip))->potato.banana =
                                                          *(char **) &date;
line 3   return tort(&(spinach->parsnip[2]), date.banana[4]) + 10;
   }
```

Generate code for the entire **`casserole`** function. Be clear about what assembly code corresponds to what line. You have this and the next page for your work.

Problem 1 Continued

Problem 2: `hashset`, Take II

You've already implemented the generic **`hashset`** once, but now you're going to do it again.

Many **`hashset`** implementations (including that used with Assignment 3) use **external** probing, so each bucket is actually a data structure storing all of the values that hash to the same value.

For this version, you're going to use quadratic **internal** probing, which means that each buckets stores at most one value. If the **`hashset`** wants to store a second value in a particular bucket, it simply can't. It needs to look somewhere else.

When entering a value, the value is hashed and reduced to a slot number i. If that particular bucket is empty, then the value is assigned and that's that. If the bucket is occupied, but the new element matches the element that resides there, the old one is replaced with the new one. If the bucket is occupied, but occupied by an element different from the one being inserted, the search for an unoccupied bucket advances to examine slot i + 1, and if that fails, i + 3, and if that fails, i + 6, then i + 10, then i + 15, and so forth. (Of course, all of these numbers—i, i + 1, i + 3, etc—are modulo the number of buckets.)

You might ask why we're going with the triangular numbers—0, 1, 3, 6, 10, 15, 21, etc—for offsets.[1] We could try i, then i + 1, then i + 2, then i + 3, etc. But quadratic probing, using the triangular number offsets, does a better job of distributing the elements across the full range of buckets. And as long as the number of buckets is a power of 2, this quadratic probing method will explore every bucket if necessary.

Of course, there's the danger that we'll run out of buckets! After all, there are a limited number of buckets, and each one can accommodate at most one user element. We're going to adopt the strategy that, at the beginning of each insertion request, we'll check to see if strictly more than 75% of the buckets are full, and if so, we'll double the number of buckets, and then rehash all previously inserted elements to their new home as if they're being inserted for the very first time.

We want to implement a C **`hashset`** generic using the above ideas. We're going to use the following **`struct`** definition for the **`hashset`**:

```
typedef int (*HashSetHashFunction)(const void *elem, int numBuckets);
typedef int (*HashSetCompareFunction)(const void *elem1, const void *elem2);
typedef int (*HashSetFreeFunction)(void *elem);

typedef struct {
   void *elems;         // pointer to dynamically allocated space
   int elemsize;        // the size of the client elements in bytes
   int count;           // the effective size of the hashset
   int alloclength;     // the allocated size of the hashset
   HashSetHashFunction hashfn;
   HashSetCompareFunction cmpfn;
   HashSetFreeFunction freefn;
} hashset;
```

The **`elems`** field stores the address of a dynamically allocated block of memory capable of storing **`alloclength`** entries, where each entry includes just enough space for a **`bool`** followed by a region large enough to store a user

[1] Note how the numbers in the sequence increase: 0 to 1 is 1, 1 to 3 is 2, 3 to 6 is 3, 6 to 10 is 4, etc.

element. The **bool** is set to **true** if and only if the bucket is occupied, in which case the remaining bytes store a shallow copy of some client element. Here's a picture that demonstrates how the bytes of a single bucket are laid out:

The **elems** field points to a dynamically allocated block capable of storing **alloclength** of these, size by side.

a. (3 points) Write a function called **HashSetNew**, which takes the address of a raw **hashset** and constructs it to represent an empty **hashset** with space for 64 client elements of the specified size. (You needn't worry about calling **assert** anywhere. Assume all incoming parameter values are good ones.)

```
/**
 * Initializes the raw space addressed by hs so that
 * it represents an empty hashset otherwise capable
 * of storing up to 64 client elements of the specified
 * size.  The specified hash, compare, and free functions
 * are used to guide insertion and manage deletion.
 *
 * @param hs the address of the raw hashset being initialized.
 * @param elemsize the size of the client elements being store (in bytes).
 * @param hashfn the generic hash function that hashed and reduces client
 *               elements to some number in the range [0, alloclength).
 * @param cmpfn the generic comparison function that returns a negative, zero,
 *              or positive value, depending on whether the first element is
 *              less than, equal to, or greater than the second element.
 * @param freefn the function that should be applied to any element ever
 *               replaced or deleted.  If there aren't any freeing needs,
 *               then freefn should be set equal to NULL.
 */

void HashSetNew(hashset *hs, int elemsize,
                HashSetHashFunction hashfn,
                HashSetCompareFunction cmpfn,
                HashSetFreeFunction freefn)
{
```

b. (7 points) Now implement **HashSetEnter**, which manages to copy the element address by **elem** into the **hashset** addressed by **hs**. It uses the quadratic internal probing technique, as described above, to find a home for the new element. **HashSetEnter** returns **true** if the new element gets inserted into a previously unoccupied bucket, and **false** if the new element replaces a previously inserted one. (Don't worry about rehashing the **hashset** if more then three quarters of the buckets are occupied. You'll worry about that in part c.) Use this and the next page for your work. Don't worry about alignment restrictions.

```
/**
 * Enters the element address by elem into the hashset
 * addressed by hs.  As with your Assignment 3 vector and
 * hashset, this particular hashset makes shallow copies
 * of the elements whenever they're being inserted for the
 * first time.
 *
 * If the new element matches some previously inserted element,
 * then the old one is disposed of and the new one is copied
 * in.  Otherwise, the new element is dropped into a previously
 * unassigned bucket, and the logical size of hashset increases by
 * one.
 *
 * @param hs the address of the hashset being updated.
 * @param elem the address of the elemsize-byte figure being
 *             inserted into the addressed hashset.
 * @return true if the new element is copied into a previously
 *              unoccupied bucket, and false if it replaces
 *              a previously inserted one.
 */

bool HashSetEnter(hashset *hs, void *elem)
{
   if (hs->count > (3 * hs->alloclength / 4))
      HashSetRehash(hs); // you'll implement this function for part c

   // complete the implementation below
```

(More space for part b.)

c. (5 points) Finally, implement the **HashSetRehash** function, which updates the **hashset** addressed by **hs** so that it has twice as many buckets and all of the elements are rehashed to take up residence in a bucket where they could have resided had the new number of buckets been the number of buckets all along. (You'll benefit by figuring out how to call **HashSetEnter** to help with the redistribution.)

```
/**
 * Doubles the number of buckets held by the addressed hashset,
 * and redistributes all of the elements.
 *
 * @param hs the address of the hashset being resized.
 */

static void HashSetRehash(hashset *hs)
{
```

(More space for part c.)

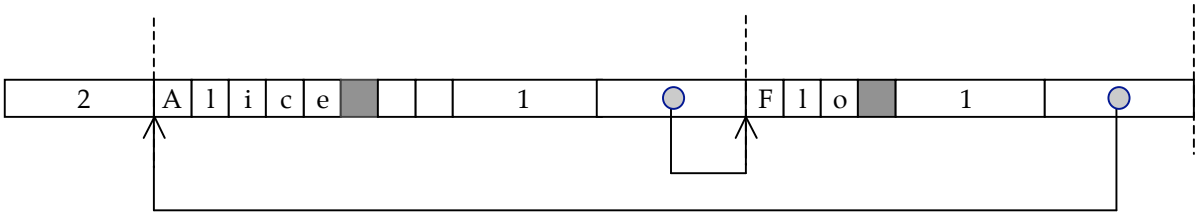
Problem 3: Decompressing Compressed Friend Graphs (10 points)

Consider the following **`struct`** definition:

```
typedef struct {
   char *name;       // dynamically allocated C string storing a person's name
   char **friends;   // dynamic array of C strings, storing friends' names
   int numfriends;   // length of the friends array
} person;
```

Your job is to implement the **`decompress`** function, which takes the base address of a single, packed binary image of friendship information (described below) and synthesizes an array of person records storing the same exact information.

Assume that **`"Alice"`** and **`"Flo"`** are the only two people in the world, and (fortunately) they're friends. The binary image storing this friendship information might look like this:



The first four bytes store the number of variably sized records that follow. Each record inlines the name of the person as a null-terminated C string, padding it with extra bytes so that the total number of bytes set aside for the name is always a multiple of 4 (In the drawing above, the shaded squares represent **`'\0'`** characters, and the empty squares can contain anything at all.) After the inlined name comes a four-byte integer, which stores the number of inlined pointers that follow. Each of those pointers points to the leading character of some other record storing information about one of his her friends. The next packed record follows, and lays out its information according to the same protocol. (A slightly more elaborate drawing is attached to the end of the exam.)

Your job is to implement the **decompress** function, which accepts the address of a binary image like the one described above, and returns the address of a dynamically allocated array of **`person`** records, where each **`person`** is populated with deep copies of a person's name and the names of all of his or her friends. The **`friends`** array within each **`person`** record is dynamically allocated to store the correct number of **`char *`**s, and each of those **`char *`**s points to dynamically allocated C strings.

Place the implementation of your **`decompress`** function on the next page. Feel free to rip this page out so you can refer to the diagram more easily.

```
/**
 * Accepts the address of the full data image storing
 * all of the friendship information, and constructs
 * and returns a dynamically allocated array of person
 * structs storing exactly the same information.
 *
 * @param image the base address of the entire data image,
 *              as described on the previous page.
 * @return the address of a dynamically allocated array
 *         of person records, where each record stores
 *         all of the friendship information about one
 *         person in the original data image.
 */

person *decompress(void *image)
{
```

Here's an example of a data image containing information about Alice, Flo, and Penny. Alice and Flo are friends, and Flo and Penny are friends, but Alice and Penny are **not** friends.