CS107 Midterm Solution

Problem 1: Your Friendly Green Grocer (15 points: 5, 4, and 6)

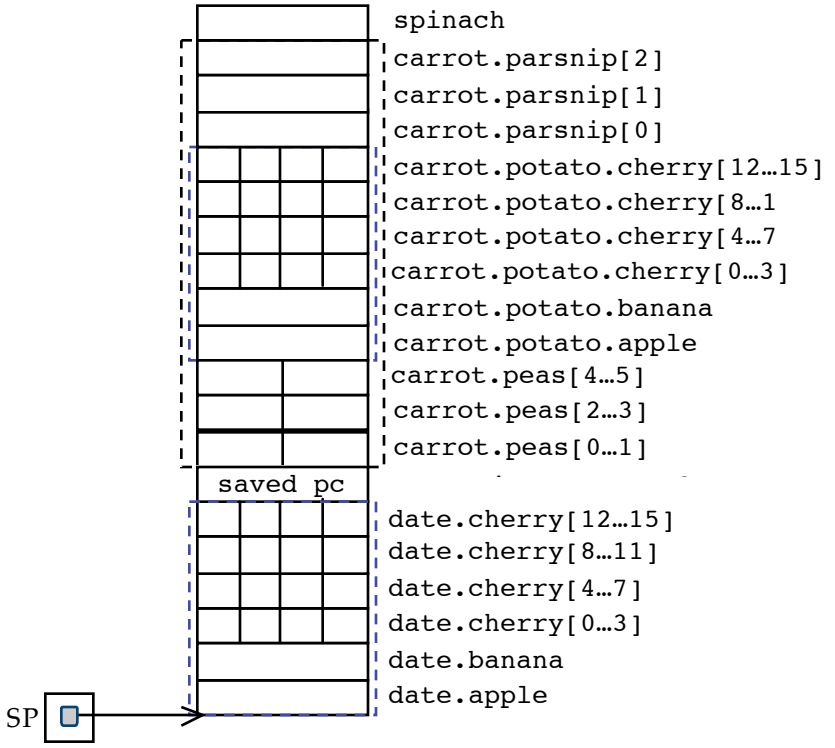
Consider the following **`struct`** definitions:

```
   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.



```
// allocation of date
SP = SP - 24;

// line 1
R1 = M[SP + 40];        // load carrot.potato.apple
R2 = R1 * 2;            // scale by sizeof(short)
R3 = M[SP + 76];        // load spinach, which is also &spinach->peas[0]
R4 = R3 + R2;           // compute address of short identified on rhs
R5 =.2 M[R4];           // load that short
M[SP + 12] =.1 R5       // populate date.cherry[4] with one-byte version
```

Criteria for Line 1 (5 points)

- Loads **`carrot.potato.apple`** and **`spinach`** using the correct offsets: 1 point
- Properly scales the **`carrot.potato.apple`** value by **`sizeof(short)`** == 2: 1 point
- Properly computes the address of the right hand side r-value: 1 point
- Loads relevant short into a register using **`.2`**: 1 point
- Stores one-byte version of that short to **`date.cherry[4]`**: 1 point

```
// line 2
R1 = M[SP];             // load the pretend char * overlaying date.apple
R2 = M[SP + 64];        // load carrot.parsnip[0]
M[R2 + 52] = R1;        // drop R1 in make-believe banana of make-believe veggie
```

Criteria for Line 2 (4 points)

- Loads the right hand side value properly with the correct number of loads: 1 point
- The sum of the offsets in whatever solution they provide add up to 116: 1 point
- The proper number of loads and stores are used to update memory as it should be: 2 points (all or nothing, because this is super important)

```
// line 3
R1 = M[SP + 76];        // load spinach again
R2 = R1 + 44;           // load address of parsnip[2] within record addressed by R1
R3 = M[SP + 4];         // load date.banana
R4 =.1 M[R3 + 4];       // load date.banana[4]
SP = SP – 8;            // make space for function call
M[SP] = R2;             // write down param 0 value
M[SP + 4] = R4;         // write down param 1 value
CALL <tort>             // jump, except RV to be populated with fruit *
SP = SP + 8;            // clean up params, expect RV to have fruit *
RV = RV + 240;          // advance RV by 10 * sizeof(fruit)
```

Criteri for Line 3 (6 points)

- 1 point for the proper number of dereferences used in synthesizing param values: point
- 1 point for the correct offsets (or sum of offsets if they lose the first point): 1 point
- Handles the **`char`** -> **`int`** conversion: 1 point
- Sets up the parameters correctly: 1 point
- Allocates and deallocates space for the params as well: 1 point
- Updates the RV by updating the current value to be 240 more.

```
// deallocation of date
SP = SP + 24;
RET;
```

Problem 2: `hashset`, Take II

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.)

```
#define kInitAllocation 64
#define EntrySize(elemsize) (sizeof(bool) + elemsize)
#define NthEntry(base, n, elemsize) ((char *)base + n * EntrySize(elemsize))

void HashSetNew(hashset *hs, int elemsize,
                HashSetHashFunction hashfn,
                HashSetCompareFunction cmpfn,
                HashSetFreeFunction freefn)
{
   hs->elems = malloc(kInitAllocation * EntrySize(elemsize));
   hs->count = 0;
   hs->alloclength = kInitAllocation;
   hs->elemsize = elemsize;
   hs->hashfn = hashfn;
   hs->cmpfn = cmpfn;
   hs->freefn = freefn;

   for (int i = 0; i < kInitAllocation; i++)
      *(bool *)NthEntry(hs->elems, i, elemsize) = false;
}
```

Criteria for Problem 2a (3 points)

- Properly calls **`malloc`**, passing in the correct number of bytes: 1 point
- Properly initializes all seven fields: 1 point
- Properly computes the addresses of all the buckets and lays down a **`false`** in the first **`sizeof(bool)`** bytes of each (or they use **`calloc`** instead of **`malloc`** to zero everything out): 1 point

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.

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

   int hashcode = hs->hashfn(elem, hs->alloclength);
   int delta = 0;

   while (true) {
      hashcode += delta;
      int bucket = hashcode % hs->alloclength;
      bool *occupied =
         (bool *) NthEntry(hs->elems, bucket, hs->elemsize);

      void *targetAddr = occupied + 1;
      if (!*occupied) { // it goes here, and we return true
         *occupied = true;
         memcpy(targetAddr, elem, hs->elemsize);
         hs->count++;
         return true;
      } else if (hs->cmpfn(elem, targetAddr) == 0) {
         if (hs->freefn != NULL) hs->freefn(targetAddr);
         memcpy(targetAddr, elem, hs->elemsize);
         return false;
      }
      delta++;
   }
}
```

Criteria for Problem 2b (7 points)

- Properly invokes the hash function to establish a base hash code for the entire enter process: 1 point
- Manages to generate the sequence of bucket numbers using the triangular number sequence, all modulo **hs->alloclength**: 1 point
- Properly computes the address of the bucket to be examined for any given iteration: 1 point (it's okay to double ding if they manually computed the address from scratch, but not triple ding. If they wrote a function or macro like I did, then don't even double ding.)
- Properly dispatches between the three insertion scenarios: bucket unoccupied versus bucket occupied by identical element versus bucket occupied by different element: 1 point
- Calls **memcpy** or **memmove** properly: 1 point
- Conditionally frees the old element if **hs->freefn** is non-**NULL**: 1 point
- Increments **hs->count** when required, and returns the correct Boolean value: 1 point

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.)

```
static void HashSetRehash(hashset *hs)
{
   hashset clone;
   memcpy(&clone, hs, sizeof(hashset));

   hs->count = 0;
   hs->alloclength *= 2;
   hs->elems = malloc(hs->alloclength * EntrySize(hs->elemsize));
   for (int i = 0; i < hs->alloclength; i++)
      *(bool *)NthEntry(hs->elems, i, hs->elemsize) = false;

   for (int i = 0; i < clone.alloclength; i++) {
      bool *occupied =
         (bool *) NthEntry(clone.elems, i, hs->elemsize);
      if (*occupied) {
         void *elem = occupied + 1;
         HashSetEnter(hs, elem);
      }
   }

   free(clone.elems);
}
```

Criteria for Problem 2c (5 points)

There are several approaches to this problem that will work just fine. My approach is a little more clever than I'd expect from anyone coding by hand in a timed situation. But the overall effect of whatever you wrote needs to be the same.

- Properly allocates a new block of memory twice as big, and updates the **alloclength** field to be twice as large: 1 point
- Manually manages the reallocation using **malloc** instead of **realloc**: 1 point
- Properly rehashes and copies over all client elements from the old block to the new one: 2 points (don't deduct faulty pointer arithmetic here unless they did it differently than they did for parts a and b)
  - Properly visits each and every bucket in the old figure: 1 point
  - Conditionally rehashes and copies over, ideally using **HashSetEnter** to do so: 1 point
- Frees the old block when everything's been copied over: 1 point

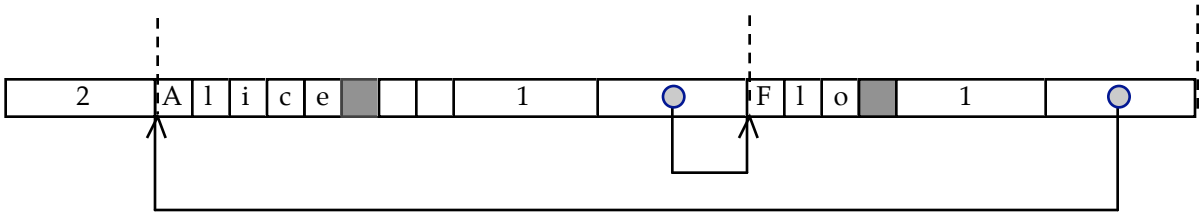
Problem 3: Decompressing Compressed Friend Graphs (10 points)

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)
{
   int numPeople = *(int *)image;
   person *people = malloc(numPeople * sizeof(person));

   int offset = sizeof(int);
   for (int i = 0; i < numPeople; i++) {
      char *name = (char *) image + offset;
      people[i].name = strdup(name);
      offset += strlen(name) + 1;
      while (offset % sizeof(char *) > 0) offset++;
      people[i].numfriends = *(int *)((char *) image + offset);
      people[i].friends = malloc(people[i].numfriends * sizeof(char *));
      offset += sizeof(int);
      char **friends = (char **)((char *) image + offset);
      for (int j = 0; j < people[i].numfriends; j++) {
         people[i].friends[j] = strdup(friends[j]);
      }
      offset += people[i].numfriends * sizeof(char *);
   }

   return people;
}
```

Criteria for Problem 3 (10 points)

- Properly extracts the total number of people: 1 point (0 points if cast is incorrect, deduct at most 2 points across entire problem for cast-related issues)
- Properly allocates a person array of just the right size: 1 point

- Clearly understands the need to maintain an offset or some construct so that it knows where each inlined record begins: 1 point
- Manages to properly maintain this offset for the lifetime of execution: 2 points (1 point dedicated specification to the padding issue between the name and number of friends)

- Fins the embedded name and makes a **`strdup`** or **`malloc`**/**`strcpy`** version of it and plants it in the name field: 1 point
- Properly extracts the number of friends: 1 point
- Properly allocations space for that many **`char *`**s and places it in the friends field: 1 point
- Uses the proper **`(char **)`** casting to establish a base address for all the friend **`char *`**s: 1 point
- Places a deep copy of the each name in one of the **`char *`** slots in the array: 1 point