- should not free the nodes of the original list.
- should be written in straight C, using no C++ whatsoever.
- should return the base address of the entire figure, expressed as an **`int *`**.
- should properly handle the empty list.
- needn't perform any error checking of any sort.

Relevant function prototypes:

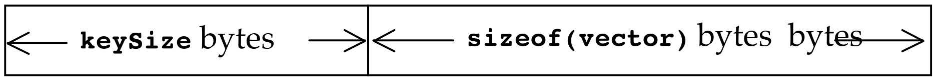
- **`strlen(const char *str)`**
  The **`strlen`** function returns the number of bytes in **`str`**, not including the terminating null character.
- **`strcpy(char *destination, const char *source);`**
  The **`strcpy`** function copies string **`source`** to **`destination`**, including the terminating null character, stopping after the null character has been copied.

```
int *serializeList(const void *list);
```

Problem 3: The `multitable`

The **`multitable`** allows a client to associate keys (of any type) with one or more values (of any type). It operates somewhat like the C++ **`map`** class, except that it's written in C and it allows multiple values to be bound to a single key.

The **`multitable`** shouldn't re-implement the **`hashset`** and the **`vector`**, but instead should be layered on top of them. A single key's collection of values should be stored in a C **`vector`**, and each key/**`vector`**-of-values pair will be stored in a C **`hashset`**. The pair itself is a manually managed chunk of memory, the size being determined by the size of the key and the size of a **`vector`**.



I've designed the **`multitable struct`** for you, but you'll be implementing three functions to demonstrate your understanding of all the low-level C functions we've been studying. Here's the reduced **`.h`** file outlining the signatures of those three functions.

```
typedef int (*MultiTableHashFunction)(const void *keyAddr, int numBuckets);
typedef int (*MultiTableCompareFunction)(const void *keyAddr1,
                                         const void *keyAddr2);
typedef void (*MultiTableMapFunction)(void *keyAddr, void *valueAddr,
                                      void *auxData);

typedef struct {
   hashset mappings;
   int keySize;
   int valueSize;
} multitable;
```

```
   void MultiTableNew(multitable *mt, int keySizeInBytes, int valueSizeInBytes,
                      int numBuckets, MultiTableHashFunction hash,
                      MultiTableCompareFunction compare);
   void MultiTableEnter(multitable *mt, const void *keyAddr, const void *valueAddr);
   void MultiTableMap(multitable *mt, MultiTableMapFunction map, void *auxData);
```

Some constraints and clarifications:
- You needn't worry about alignment restrictions.
- The bytes making up a key must be laid out and replicated according to the diagram above.
- The bytes making up a value must be replicated as elements of the vector of values.
- **`MultiTableHashFunction`**s know how to hash keys, not key/**`vector`** pairs. The keys guide the search through the embedded **`hashset`**.
- **`MultiTableCompareFunction`**s know how to compare keys, not key/**`vector`** pairs. Again, it's the keys that guide the search.
- We don't worry about freeing at all, so just pass in **`NULL`** whenever you're required to provide a **`VectorFreeFunction`** or a **`HashSetFreeFunction`**.

a) First implement the **`MultiTableNew`** function. This should be very short, and it should be relatively painless provided you read the header comments (which contain some implementation specifications as well.)

```
typedef struct {
   hashset mappings;
   int keySize;
   int valueSize;
} multitable;

/**
 * Function: MultiTableNew
 * -----------------------
 * Initializes the raw space addressed by mt to be an empty
 * multitable otherwise capable of storing keys and values of
 * the specified sizes.  The numBuckets, hash, and compare parameters
 * are supplied with the understanding that they will simply be passed
 * to HashSetNew, as the interface clearly advertises that a hashset
 * is used.  You should otherwise interact with the hashset (and any
 * vectors) using only functions which have the authority to manipulate
 * them.
 */

typedef int (*MultiTableHashFunction)(const void *keyAddr, int numBuckets);
typedef int (*MultiTableCompareFunction)(const void *keyAddr1,
                                         const void *keyAddr2);

void MultiTableNew(multitable *mt, int keySizeInBytes, int valueSizeInBytes,
                   int numBuckets, MultiTableHashFunction hash,
                   MultiTableCompareFunction compare);
```

b) Next, implement the more difficult **`MultiTableEnter`**, which ensures the value at the specified address is included in the vector of values bound to the specified key (also specified via its address.) Append the value to the end of the **`vector`** without searching to see if it already exists (so duplicate values are allowed). And be sure to handle the situation where the key is being inserted for the very first time (in which case you're also introducing a new key/**`vector`** pair.)

```
/**
 * Function: MultiTableEnter
 * -------------------------
 * Enters the specified key/value pair into the multitable.
 * Duplicate values are permitted, so there's no need to search
 * existing vectors for a match.  You must handle the case where
 * the key is being inserted for the very first time.
 * Understand that the patterns for each key and value are replicated
 * behind the scenes (using memcpy/memmove/VectorAppend as needed).
 */

void MultiTableEnter(multitable *mt, const void *keyAddr, const void *valueAddr);
```

c) Finally, implement the **`MultiTableMap`** routine. **`MultiTableMap`** applies the specified mapping function to each key/value pair. You'll certainly need to use **`HashSetMap`** to reach all key/value pairs, but you may iterate over all value **`vector`**s using **`VectorLength`**, **`VectorNth`**, and a **`for`** loop. You will need to write a helper function and define a helper **`struct`** in order to get this to work. (Note that the **`MultiTableMapFunction`** and the **`HashSetMapFunction`** are incompatible types.)

```
/**
 * Function: MultiTableMap
 * -----------------------
 * Applies the specified MultiTableMapFunction to each key/value pair
 * stored inside the specified multitable.  The auxData parameter
 * is ultimately channeled in as the third parameter to every single
 * invocation of the MultiTableMapFunction.  Just to be clear, a
 * multitable with seven keys, where each key is associated with
 * three different values, would prompt MultiTableMap to invoke the
 * specified MultiTableMapFunction twenty-one times.
 */

typedef void (*MultiTableMapFunction)(const void *keyAddr,
                                      void *valueAddr, void *auxData);

void MultiTableMap(multitable *mt, MultiTableMapFunction map, void *auxData);
```

Problem 4: Acting As `multitable` Client Code

Every post office in America has one: a `multitable` mapping zip codes to US cities. Some zip codes (like 08077, I just happen to know) identify multiple cities (Cinnaminson, NJ; Riverton, NJ; and Palmyra, NJ—small towns don't need their own zip code), so a `multitable` makes sense. This `multitable` sets aside exactly six bytes for the zip code keys, since all zip codes can be stored as static, null-terminated character arrays. But the city names can be arbitrary long, so it's best to store those as pointers to dynamically allocated character arrays (also null-terminated, of course). The `multitable` that's relevant to us would be initialized like this (the details of `ZipCodeHash` and `ZipCodeCompare` are irrelevant—just assume they do the right thing):

```
multitable zipCodeDatabase;
MultiTableNew(&zipCodeDatabase, 6 * sizeof(char), sizeof(char *),
              100000, ZipCodeHash, ZipCodeCompare);
```

`ListRecordsInRange` is designed to map over such a `multitable` and print all records whose zip code lies in the specified range. Your job is to complete the implementation for the `InRangePrint` mapping function, which reinterprets all of the `void *` parameters to be the true types they really are, and then prints the record if and only if the zip code of the entry is greater than or equal to the low endpoint and less than or equal to the high endpoint. If the entry falls out of range, then `InRangePrint` prints absolutely nothing. I've taken care of the variable declarations and the conditional print, but you should fill the space with the code that properly initializes the four local variables I declare for you. If you get that right, then the `strcmp` and `printf` calls do what needs to be done.

```
void ListRecordsInRange(multitable *zipCodes, char *low, char *high)
{
   char *endpoints[] = {low, high};
   MultiTableMap(zipCodes, InRangePrint, endpoints);
}

static void InRangePrint(void *keyAddr, void *valueAddr, void *auxData)
{
   char *zipcode;
   char *city;
   char *low;
   char *high;
```

include code that properly initializes the four local variables to be what they need to be so that the range check and the `printf` statement work properly.

```
   if ((strcmp(zipcode, low) >= 0) && (strcmp(zipcode, high) <= 0)
      printf("%5s: %s\n", zipcode, city);
}
```