# Section Handout

**Problem 1: The `sparsestringarray`**

A **`sparsestringarray`** is an array-like data structure that provides constant time access to its elements, and constant time insertion and deletion. It layers array semantics over an ordered collection of C strings, with the understanding that most of the strings are empty.

The **`sparsestringarray`** is different from C arrays and other array-like data structures, because it's aggressively miserly in its use of memory. Each empty string requires just one bit of storage, which is less than 3% of the memory cost incurred by the allocation of a full **`char *`**. The implementation is slower than true C arrays and our **`vector`** from Assignment 3, but it's a wise choice when memory is at a premium and an overwhelmingly large fraction of the strings being stored are the empty string.

Our **`sparsestringarray`** is backed by an array of groups, where each group is responsible for managing a contiguous subset of array indices. The programmer specifies not only the logical length of the **`sparsestringarray`**, but also the group size. If the logical length of the full **`sparsestringarray`** is, for instance, established as 50,000, and the group size is established to be 100, then group 0 would manage indices 0 through 99, group 1 would manage indices 100 through 199, group 2 would manage indices 200 through 299, and so forth. All search, insert, and delete operations are passed on to the appropriate group.

```
typedef struct {
   group *groups;    // dynamically allocated array of structs, defined below
   int numGroups;    // number of groups
   int arrayLength;  // logical length of the full sparsestringarray
   int groupSize;    // number of strings managed by each group
} sparsestringarray;
```

Each group contains a bitmap, which is an array of **`bool`**s whose length is equal to the group size, and a C **`vector`** (yes, Assignment 3's **`vector`**) to store nonempty, dynamically allocated C strings. Search for a particular element amounts to search within a particular group at index **`i`**. If **`bitmap[i]`** is **`false`**, then the string at the **`i`**$^{th}$ position is understood to be the empty string. If instead **`bitmap[i]`** is **`true`**, then the group needs to find the corresponding string in the C **`vector`**.

```
typedef struct  {
   bool *bitmap;     // set to be of size 'groupSize'
   vector strings;   // vector of dynamically allocated, nonempty C strings
} group;
```

Each **`true`** in a group's bitmap corresponds to some **`char *`** (addressing a dynamically allocated C string) in the same group's **`vector`**. The **`true`** at the lowest index in the bitmap corresponds to the 0$^{th}$ entry in the **`vector`**; the **`true`** at the second lowest index in the bitmap

corresponds to the 1st entry in the **vector**, and so forth; the total number of **true**s should be equal to the logical length of the accompanying **vector**. (In practice, the **bool** array would be compressed to use just one bit of memory for each Boolean value, but for our purposes we won't implement that optimization, since it requires advanced C directives not covered in CS107.)

Here are the prototypes of the four functions you'll be implementing:

```
void SSANew(sparsestringarray *ssa, int arrayLength, int groupSize);
bool SSAInsert(sparsestringarray *ssa, int index, const char *str);
void SSAMap(sparsestringarray *ssa, SSAMapFunction mapfn, void *auxData);
void SSADispose(sparsestringarray *ssa);
```

Of course, the **sparsestringarray** gives the illusion that all strings, both empty and nonempty, are stored in an array-like manner. You know as the implementer the internal representation is such that only nonempty strings are really stored. Your job is to implement these four functions in a way that's consistent with the description outlined on the previous page. Here's a test program that illustrates how a client can interact with a **sparsestringarray**.

```
static void CountEmptyPrintNonEmpty(int index, const char *str, void *auxData)
{
   if (strcmp(str, "") != 0) {
      printf("Oooo! Nonempty string at index %d: \"%s\"\n", index, str);
   } else {
      (*(int *)auxData)++;
   }
}

int main(int argc, char **argv)
{
   sparsestringarray ssa;
   SparseStringArrayNew(&ssa, 70000, 35);

   SparseStringArrayInsert(&ssa, 33001, "need");
   SparseStringArrayInsert(&ssa, 58291, "more");
   SparseStringArrayInsert(&ssa, 33000, "Eye");
   SparseStringArrayInsert(&ssa, 33000, "I");
   SparseStringArrayInsert(&ssa, 67899, "cowbell");

   int numEmptyStrings = 0;
   SparseStringArrayMap(&ssa, CountEmptyPrintNonEmpty, &numEmptyStrings);
   printf("%d of the strings were empty strings.\n", numEmptyStrings);

   SparseStringArrayDispose(&ssa);
   return 0;
}
```

Here's the output of that test program:

```
Oooo! Nonempty string at index 33000: "I"
Oooo! Nonempty string at index 33001: "need"
Oooo! Nonempty string at index 58291: "more"
Oooo! Nonempty string at index 67899: "cowbell"
69996 of the strings were empty strings.
```

a) Present implementations for **SSANew** and **SSADispose**. **SSANew** is designed to take the address of a raw **sparsestringarray** record and construct it to represent an array of the specified logical length, where every single element is the empty string. **SSADispose** takes the address of a previously constructed **sparsestringarray** and disposes of all resources that have been tapped during its lifetime. (Note that all **sparsestringarray**s own deep copies of all their nonempty strings.)

```
/**
 * Function: SSANew
 * ----------------
 * Constructs the sparsestringarray addressed by the first argument to
 * be of the specified length, using the specified group size to decide
 * how many groups should be used to back the implementation.  You can
 * assume that arrayLength is greater than groupSize, and for simplicity you
 * can also assume that groupSize divides evenly into arrayLength.
 */

void SSANew(sparsestringarray *ssa, int arrayLength, int groupSize);

/**
 * Function: SSADispose
 * --------------------
 * Disposes of all the resources embedded within the addressed
 * sparsestringarray that have built up over the course of its
 * lifetime.
 */

void SSADispose(sparsestringarray *ssa);
```

b) Implement the more involved **SSAInsert** function, which ensures that the C string addressed by **str** is cloned and stored in the proper vector within the proper group and that the proper bit in the bitmap is set. Note that in some cases the implementation needs to call **VectorInsert** (in the case where the string is being inserted at the specified index for the very first time) and in others needs to call **VectorReplace** (because the **SSAInsert** call is overwriting some nonempty string previously inserted at the very same index. ) You may assume the string being inserted is nonempty. You should return **true** if a string is being inserted at the specified position for the very first time, false otherwise.

```
/**
 * Function: SSAInsert
 * -------------------
 * Inserts the C string addressed by str into the sparsestringarray addressed
 * by ssa at the specified index.  If some nonempty string already resides
 * at the specified index, then it is replaced with the new one.  Note that
 * SSAInsert makes a deep copy of the string address by str.
 */

bool SSAInsert(sparsestringarray *ssa, int index, const char *str);
```

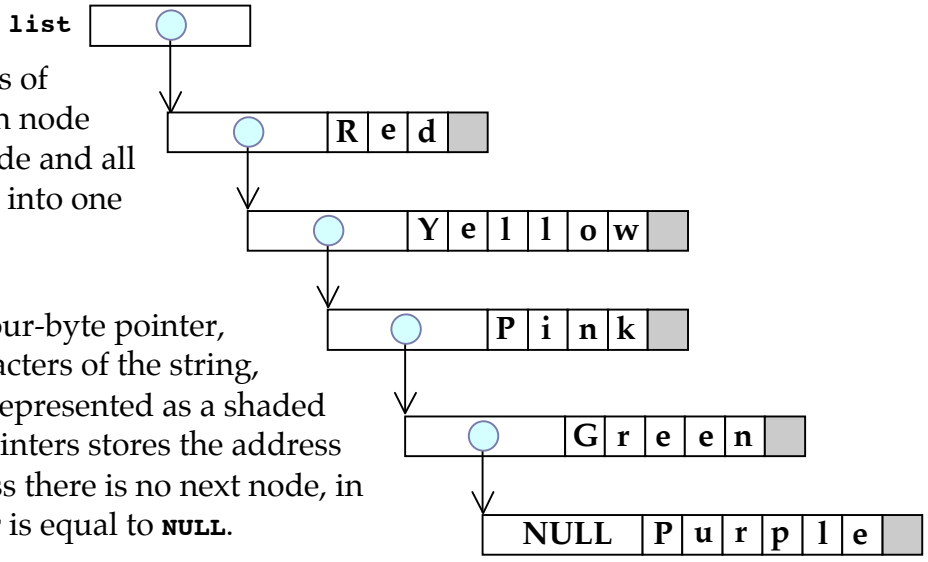
c) Finally, implement the **SSAMap** routine, which applies the specified mapping function to every single index/string pair held by the specified **sparsestringarray**. Note that the mapping function is called on behalf of all strings, empty and nonempty. The specified auxiliary data is channeled through as the third argument to every single call.

```
/**
 * Function: SSAMap
 * ----------------
 * Applies the specified mapping routine to every single index/string pair
 * (along with the specified auxiliary data).  Note that the mapping routine
 * is called on behalf of all strings, both empty and nonempty.
 */

typedef void (*SSAMapFunction)(int index, const char *str, void *auxData)
void SSAMap(sparsestringarray *ssa, SSAMapFunction mapfn, void *auxData);
```

**Problem 2: Serializing Lists of Packed Character Nodes**

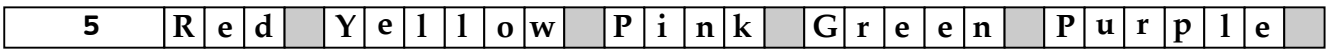
Write a function **serializeList** to convert a linked list to a single stream of null-delimited characters arrays.




The linked list consists of a series of variably sized nodes, where each node packs the address of the next node and all of the characters of a C string all into one contiguous block.

Notice that each node stores a four-byte pointer, followed by the individual characters of the string, followed by the null character (represented as a shaded box). Each of these four-byte pointers stores the address of the next node in the list, unless there is no next node, in which case the four-byte pointer is equal to **NULL**.

**serializeList** synthesizes a dynamically allocated serialization of such a list. The serialization starts off with a **sizeof(int)**-byte figure storing the number of C strings. The serialization then continues with each of the C strings laid down side by side, one after another in their original order. The individual strings are separated by the null characters, and the final string in the character array is null-terminated as well. If handling the above list, **serializeList** would build and return the base address of the **int** storing the **5**:




**serializeList** takes a **const void *** and constructs the corresponding serialization. Your implementation:

- should be implemented iteratively in one single pass over the list.
- should create a serialization using the exact number of bytes needed.

- 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:

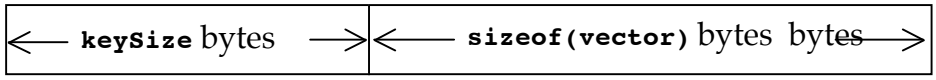
- **`strlen(const char *str)`**
  The **`strlen`** function returns the number of bytes in **`str`**, not including the terminating null character.
- **`strcpy(char *destination, const char *source);`**
  The **`strcpy`** function copies string **`source`** to **`destination`**, including the terminating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

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

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.