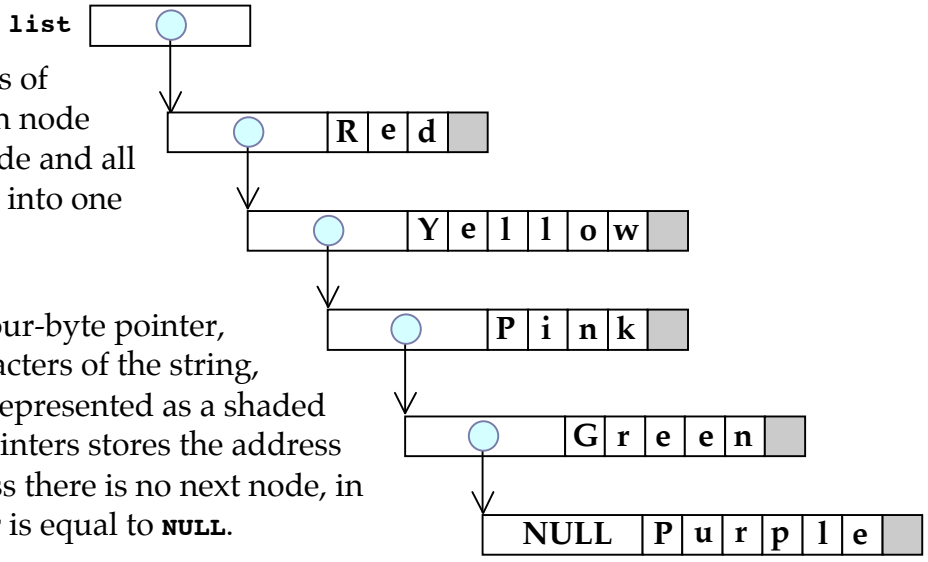
c) Finally, implement the **SSAMap** routine, which applies the specified mapping function to every single index/string pair held by the specified **sparsestringarray**. Note that the mapping function is called on behalf of all strings, empty and nonempty. The specified auxiliary data is channeled through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

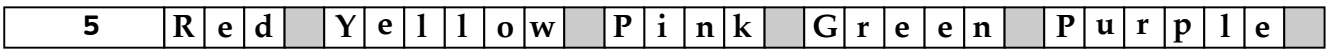
Write a function **serializeList** to convert a linked list to a single stream of null-delimited characters arrays.



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:

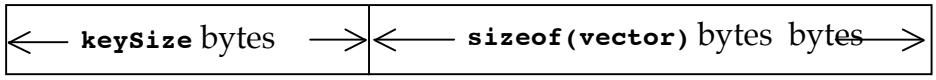
- **`strlen(const char *str)`**
  The **`strlen`** function returns the number of bytes in **`str`**, not including the terminating null character.
- **`strcpy(char *destination, const char *source);`**
  The **`strcpy`** function copies string **`source`** to **`destination`**, including the terminating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