```
#define Nodeptr struct node*       //prefer   typedef
```

See King P. 335 or K&R Section 6.7 (typedef):
create data type names

```
typedef struct node* Nodeptr;

typedef  struct node {
    int data;
    Nodeptr next;       // struct node* next;
} Node ;

Nodeptr mknode (int value)
{
    Nodeptr np;
    / /  np = (Nodeptr) malloc (sizeof (Node));
    np = (Nodeptr) malloc (sizeof (struct node));
    if (np != NULL) {
        np->data = value;
        np->next = NULL;
    }
    return np;
}
```

Sometimes we want one field in a data structure to hold two values of different type. Say for instance either an integer or a float. We can do this through the use of a union--forming something much like a variant record in Pascal.

```
typedef  struct node {
    union data {
        int  digits;
        float  number;
    }
    Nodeptr  next;        //  struct  node*  next;
} Node ;
```

Having created a sample node with a declaration like

```
    Node datanode;
```

which is the same as

```
    struct node datanode;
```

and one can either store digits with

```
    datanode.data.digits = 19;
```

or one can store a float into the same space, with

```
    datanode.data.number = 91.3;
```

We can also create a node with:

```
    Nodeptr  dataptr;
    dataptr = (Nodeptr) malloc ( sizeof Node);
```

and can set them via the pointer variable

```
    dataptr->data.digits  =  19;
```

or

```
    dataptr->data.number  =  91.3;
```

Note that what we are doing is saving a little space in the computer (not really an overriding consideration these days). The savings here would be small. Instead of needing space for both an integer (digits) and a float (number), taking 4 + 8 bytes. We can overlay them and use only Max (4, 8) = 8 bytes (space for the longer).

Use of Unions can be quite elaborate, but what is clear is that once you have stored an item into a union structure it is impossible to retrieve unless you know what type of variable was stored (in this case an integer or a float). It follows therefore that you must also store in the node some kind of a TAG that interprets uniquely the item in the "variant" field.

Typically we might do this though the use of some defined constants:

```
#define INTEGER -1
#define FLOAT 1
struct node {        // note datanode creation here
    int TAG;
    union data {
        int  digits;
        float  number;
    }
    Nodeptr next;        // struct node* next;
} datanode ;

int kind = INTEGER;      // or perhaps FLOAT

datanode.TAG = kind;
if (kind == INTEGER)
    datanode.data.digits = 19;
else if (kind == FLOAT)
    datanode.data.number = 91.3;
else printf ("Illegal tag %d", kind);
```

Once the item has been stored in datanode it can be retrieved by first interrogating the TAG field. See King P. 345-350 or K&R P. 148

What now if you wanted an array of such a variant record structure?
(As indeed you might for the second assignment!)

```
typedef struct node* Nodeptr;

typedef struct node {
    int TAG;
    union data {
        int digits;
        float number;
    }
} Node ;
```

At this stage we have only a definition of a sample element in our array. If we wanted to create M of them we would first need to declare a pointer to such an element

```
    Nodeptr np;
```

then get space for the array with

```
    np = (Nodeptr) malloc (M*sizeof(Node));
```

or

```
    np = (Nodeptr) calloc (M, sizeof(Node));
```

To access the zero'th TAG field we could write

```
    kind = np->TAG;
```

while the i'th tag:

```
kind = (np+i)->TAG;
```

and one of the union fields would be:

```
value = (np+i)->data.number;
```

provided i < M and TAG == FLOAT

An example:

Let use now review a typical C115 hashing problem. Given an array of M boxes. Into each we are going to store one animal from an input queue:
"dog", "cat", "hen", etc.

Assume we want to store them in these boxes according to some hashing scheme.

First we must compute a hash number based on some transformation of the animal type.

```
H = Transform ("dog");
```

Then we must compute the hash index modulo M to determine into which box to store the animal.

```
I = H % M;
```

If the box is empty pop the animal in.

If the box is occupied, compute the address of the next available box with:

```
I = (I +1) % M;
```

until you find an empty box. Note our method guarantees that you will eventually look in all M boxes before determining that there is no room at the Inn.

How do you compute H? What are the properties of a good Transform? Ideally you would like one that computes H so that its values are uniformly distributed across the full range of positive integers that can be stored in a 4-byte unsigned integer.

[Actually, we would really like the right-most k bits to be random, where k=4 if M is 16, etc., but this is even more difficult].

One scheme is given in K&R page 144. This looks kind of expensive (an addition and multiplication for each character in the name), but is it really?

Another scheme is to try and use the bit-wise XOR operator (^) that we featured in earlier discussions about Gray Codes and BitStrings. Too elaborate?

Or we could try some el-cheapo scheme like adding together the 8-bit integers that correspond to each character in the animal-type name. Poor, but a start.

Thus Transform ("dog") is 100+111+103 = 314
while Transform ("cat") is 99+97+116 = 312 and
Transform ("hen") is 315.

If M = 10 (ten boxes), then the dog would go in box 4, the cat in box 2 and the hen in box 5, and everyone would be happy.

Retrieving is the reverse process. If we want the "hen", look in box

```
Transform ("hen") % M;
```

That is, box 5 and take the hen.

If you want a "rat", compute H = 114+97+116 = 312 and then look in box 312 % 10 = 2.

There you will find a "cat", not a "rat" so look in box (2+1) % 10 = 3, which is empty.

Therefore we have no rats here.

A possible structure to support this menagerie is:

```
#define M 10

typedef struct box* Boxptr;

typedef struct box {
    char* s;
} Home;
```

If we wanted an array of these we would declare:

```
    char* animal;
    int I;
    Boxptr p;
```

and set

```
    p = (Boxptr) malloc (M*sizeof(Home));
```

and compute

```
    I = Trans("dog") % M;
```

or perhaps

```
    scanf ("%s", &animal);
    I = Trans(animal) % M;
```

and to put the animal in its new home with:

```
    (p+I)->s = animal;
```

I hope this introduction provides some basic starting point for the assignment. It may look daunting, and it certainly will take a lot of pencil and paper planning (and lots of debugging time fixing up bad pointers).

The design time will be well rewarded.

Only go near a computer to check out basic ideas that you want to use.

* Sketch out clearly model solutions for all 8 functions first.

* Clearly identify all the helper functions you are going to need (like a hash function, and procedures that can be called from any of your main functions to perform routine operations on your data structure).

* Keep in mind, as with the first assignment, you are designing a solution to a specified interface.

* On the one hand this requires great discipline, on the other it will surely keep you out of trouble by ensuring that your design has some coherent structure!

* As always your work is best spread over as long a period as possible, with considerable thought going into forming a clean and simple design.

MapGetSS (Map M, String I) -> String

MapGetSI (Map M, String I) -> int

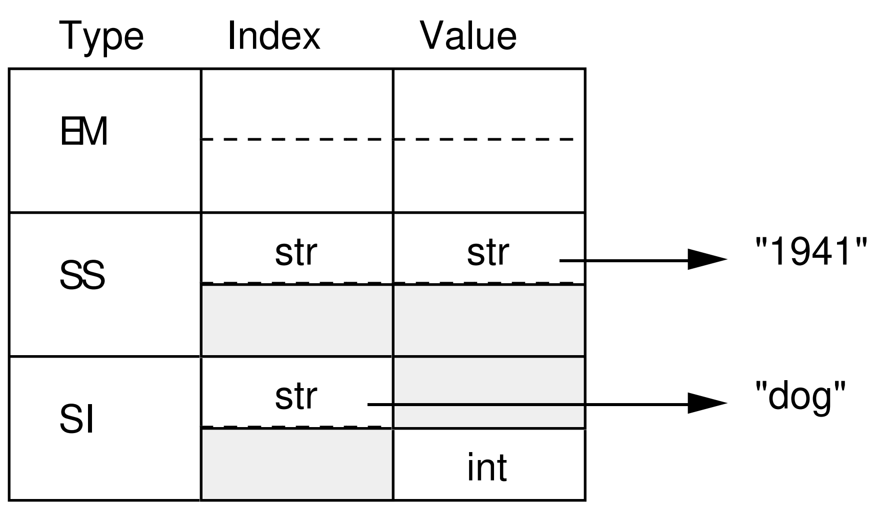
These four functions use I to locate the index-value pair < I,V > in map M and return V.

If the pair < I,V > does not exist then MapGetSS and MapGetIS quietly adds < I,"" > to M and returns the null string,

and MapGetSI and MapGetII quietly adds < I,0 > to M and returns 0.

If the type of V is String and an int is expected then the value atoi(V) is returned. *The entry was previously created by MapSetSS, perhaps. There are other ways of converting a string to an integer, but assume a valid 32-bit integer will result.*

If the type of V is int and a string is expected then V is converted to a string
----------------your choice design decision-------------------
which then replaces the old value of V (which must be properly disposed of), and V is returned.



top.c and spell.c use SI entries only, anagram.c uses SS entries