```c
#define Nodeptr struct node*      //prefer typedef
```
See King P. 335 or K&R Section 6.7 (typedef):
create data type names

```c
typedef struct node* Nodeptr;

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

Nodeptr mknode (int value)
{
    Nodeptr 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.

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

Having created a sample node with a declaration like
```c
Node datanode;
```
which is the same as
```c
struct node datanode;
```
and one can either store digits with
```c
datanode.data.digits = 19;
```
or one can store a float into the same space, with
```c
datanode.data.number = 91.3;
```


We can also create a node with:
```c
Nodeptr dataptr;
dataptr = (Nodeptr) malloc ( sizeof Node);
```
and can set them via the pointer variable
```c
dataptr->data.digits = 19;
```
or
```c
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:

```c
#define INTEGER -1
#define FLOAT 1
struct node {       // note datanode creation here not typedef
    int TAG;
    union {
        int digits;
        float number;
    } data;
    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 fprintf (stderr, "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 if you now wanted an array of such a variant record structure?
(As indeed you might for a practical application!)

```
typedef struct node* Nodeptr;

typedef struct node {
    int TAG;
    union {
        int digits;
        float number;
    } data;
} 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 box i 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?

```
    unsigned Transform (char* s) {
        unsigned hashval;
        for (hashval = 0; *s != '\0'; s++)
            hashval = *s + 31*hashval;
        return hashval;
    }
```


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

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));
    for (i=0; i < M; i++)
        (p+i)->s = NULL;
```

To find a home for a "dog" we would compute:

```
    i = Transform("dog") % M;
```

or more generally:

```
    char animal[10];
    scanf ("%s", &animal[0]);
    i = Transform(animal) % M;
```

and to put the animal in its new home with


```
    if ((p+i)->s == NULL)
        (p+i)->s = &animal[0];      // or simply (p+i)->s = animal
    else  //conflict, box is occuppied.
```

If we have a conflict we are stuck. Either we need a more elaborate structure, one that allows us to form a linked list of boxes where the overflow animals reside, or we begin some kind of sequential search in the neighbourhood of the initial entry box to find an empty one.

```
While ((p+i)->s != NULL)
    i = (i + 1) % M;
(p+i)->s = animal;
```

This introduction provides some basic starting point for hashing. 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.

Initially just use the computer to check out basic ideas that you want to try.

* Sketch out clearly model solutions for access functions.

* 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 any 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.

Placing animals in boxes is easy (just as easy as updating pointers to names of animals. But what happens when it comes time to remove animals?

Discuss the issues, come up with some strategies.