Data Structures and Algorithms (DSA) Course 4

Iulian Năstac

10. Functions for dynamic memory allocation (recapitulation)

- Dynamic allocation is a specific characteristic allowed by some computing languages, in which a program can obtain memory at runtime.

- There are two ways that memory gets allocated for data storage:
  - Compile Time (or static) Allocation
    - Memory for named variables is allocated by the compiler
    - Exact size and type of storage must be known at compile time
  - Dynamic Memory Allocation
    - dynamically allocated space usually placed in a program segment known as the *heap* or the *free store*
    - pointers are crucial for dynamic allocation

(recapitulation)

- The area of memory used for dynamic allocation is obtained from the **heap** memory area.
- C dynamic memory allocation refers to performing manual memory management for dynamic memory allocation in the C programming language via a group of functions in the C standard library, namely malloc, realloc, calloc and free.
- The main functions for dynamic allocation in C are **malloc** and **free** (which can be found in the header files, like alloc.h or malloc.h - depending on the compiler).

11. Incorrect use of pointers (recapitulation)

- Often the errors provided by pointers are very difficult to detect.
- The incorrect uses of pointers usually consist in reading or writing in an unknown area of memory.
- Major drawback is that the effects of these mistakes are only visible during program execution, and not during compilation.
- Finding the faults is often a difficult task for programmers, especially when for some applications them are not directly (or immediately) visible.

Cap. Structures

(recapitulation)

1. Defining the concept of structure

- C programming language can process single or grouped variables, which enable global processing.

- An example of the second category is the matrix, which is in fact an ordered set of data of the same type (the order of the elements is realized by indices).

- However, often it is useful to group the data other than the one used for matrices. This time the data are not necessarily of the same type and requires a global processing. This form of group is called **structure**.

Reference to elements of such groups didn't use indices but a special way that include the name of structure. Components of the groups can be groups themselves. Furthermore, it is possible to define a hierarchy of groups.

Thus:

- The group that is not part of another group is of the highest level;
- Data that didn't include other groups of data are basic (or elementary) data.

As a very general definition, we can say that the data grouped according to a hierarchy are called **structures**.

Notes:

- Basic data of a structure can be isolated (single) or matrices;
  Each structure represents a new type of data, defined by the user.

2. Declaration of structure

The general syntax for a struct declaration in C is:

```
struct tag_name
        { type member1;
          type member2;
        …
        } identification_1, identification_2, …,
identification_n;
```

Here tag_name or identification_i are optional in some contexts.

Thus:

- if identification _1, identification _2, …, identification_n are absent, then tag_name should be present.

- if tag_name is absent, then at least identification 1 should be present.

Notes:

- A variable of the structure type can be declared subsequently.

  struct tag_name identification _1, …, identification_n*;*

- A statement of a specific structure *identification_i* (where *i* = *1…n*) may be replaced by a k-dimensional array of elements of tag_name type:

  identification_i[lim1][lim2]...[limk]

Examples:

1) The following three code examples will have the same result:

```
struct calendar_data
{int day;
char month[11];
int year;
} birth_date, employment_date;
```

or

```
struct
{int day;
char month[11];
int year;
} birth_date, employment_date;
```

or

```
struct calendar_data
{int day;
char month[11];
int year;
};
...
struct calendar_data  birth_date, employment_date;
```

2) Structure containing personal information :

```
struct  personal_data
  { char name[100];
    char address[1000];
    struct calendar_data   birth_date, employment_date;
    char gender;
  };

    ..............

struct personal_data   manager, employees[1000];
```

The variable named *manager* is a structure of *personal_data* type, and *employees[1000]* is an array of structures.

3) Define complex numbers A, B and C.

```
struct COMPLEX
        {double real;
         double imag;
        }A, B, C;
```

4) The position of a point on the screen is given by two coordinates:

```
struct punct
   { int x;
     int y;
   };
   ...
struct punct poz;
```

3. Access to the elements of a structure

The access to the elements of a structure can be done in one of the following two ways:

- *struct_name.date_name*
- *pointer -> date_name*

where: *struct_name is the name of* structure;
date_name is the name of a specific component of the structure;
pointer is a pointer to that structure.

Examples:

1) struct calendar_data

```
{int day;
char month[11];
int year;
} dc,d[10];
    ...
 dc.day=1;
dc.year=2015;
strcpy(dc.month,”March”);
...
d[3].day=dc.day;
d[3].year=dc.year;
strcpy(d[3].month,dc.month);
...
```

2) Function that calculates and returns the ***modulus*** of the complex number z.

```
double modulus(COMPLEX *z)
        {
        return sqrt(z->x * z->x + z->y * z->y);
        }
```

It should be noted that the components of an structure can be **initialized**.

4. Typedef declarations

- By declaring a structure, we introduce a new type.
- In general, a name can be assigned to a type, whether it is a predefined type or one defined by the programmer. This should be done by using the following syntax:

typedef type new_type_name;

where

- *type* is a predefined type or one previously defined by the programmer;
- *new_type_name* is the name allocated to the new type.

Examples:

1) By using the statement

typedef double REAL;

the data

REAL x,y;

Are of the double type.

2) Declaring COMPLEX type.

```
typedef struct
        { double real;
          double imag;
        } COMPLEX;
...
```

We can then declare complex numbers:

COMPLEX z, tz[10];

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<conio.h>
typedef struct {
                double x;
                double y;
               } COMPLEX;
void sum_c(COMPLEX *a, COMPLEX *b, COMPLEX *c);

int main()
{
 COMPLEX a,b,c;
 printf("\n\n Enter the real and the imaginary part ");
 printf("\n of the first complex number :\n");
 if(scanf("%lf %lf",&a.x,&a.y)!=2)
        {
          printf("\nError");
          exit(1);
         }
 ….
```

```
…
printf("a = %g + i*(%g)\n",a.x,a.y);
 printf("\n\n Enter the real and the imaginary part ");
 printf("\n of the second complex number :\n");
 if(scanf("%lf %lf",&b.x,&b.y)!=2)
        {
          printf("\nError");
          exit(1);
         }
 printf("b = %g + i*(%g)\n",b.x,b.y);
 sum_c(&a,&b,&c);
 printf("\na+b = %g + i*(%g)",c.x, c.y);

 getch();
}


void sum_c(COMPLEX *a, COMPLEX *b, COMPLEX *c)
{
  c->x = a->x + b->x;
  c->y = a->y + b->y;
}
```

5. Unions

Introduction

Usually, in C a memory area is assigned according to the type of variable. Its allocated memory can keep only the data of the mentioned type.

For example:

double x;

For **x** is allocated 8 bytes (64 bits) in the computer memory in order to store a real number.

What is an union in C?

- Unions in C are related to structures and are defined as objects that may hold (at different times) objects of different types and sizes.
- They are analogous to variant records in other programming languages. Unlike structures, the components of a union all refer to the same location in the memory.
- In this way, a union can be used at various times to hold different types of objects, without the need to create a separate object for each new type.
- The size of a union is equal to the size of its largest component type.

Definition

- A **union** is a special data type available in C that enables you to store different data types in the same memory location.

- Notes:
  - You can define a union with many members, but only one member can contain a value at any given time.
  - Unions provide an efficient way of using the same memory location for multi-purpose tasks of different variables (but not in the same time).

Examples:

1)
```
union a
        {int x;         /* 2 bytes for x */
        long y;         /* 4 bytes for y */
        double r;       /* 8 bytes for r*/
        char c;         /* 1 byte for c */
        } var;
```

In the above statement ***var*** is a union of the type **a**. Accessing variables can be done with: *var.x;* or *var.y;* or *var.r;* or *var.c*; but in different locations of the program (and not at the same time).

For ***var*** it is allocated a memory area which is sufficient to keep the maximum number of bytes (8 bytes in this example). If would be replaced ***union*** with ***struct***, then it will be necessary 15 bytes.

2)

```
struct data
        { int timp;
          union { int i;
                  float f;
                  double d;
                } zc;
        } util;
```

We can access:

```
util.zc.i=123;
```

As observed, in contrast to the structure, a union can not be initialized.

6. Bit fields

- C also provides a special type of structure member known as a **bit field**, which is an integer with an explicitly specified number of bits.

- A bit field is declared as a structure member of type int, signed int, unsigned int, or boolean, following the member name, a colon (**:**) and the number of bits that it should occupy.

- The total number of bits in a single field must not exceed the total number of bits from its declared type.

Basically, more fields can be grouped to form a structure:

```
struct identification
        { field_1;
          field_2;
          ...
          field_n;
        } name_1, name_2, ..., name_n;
```

Syntax of bit field:

type name_field: length_in_bits;

or:

type : length_in_bits;

where type can be int, signed int, unsigned int, or boolean.

Notes:

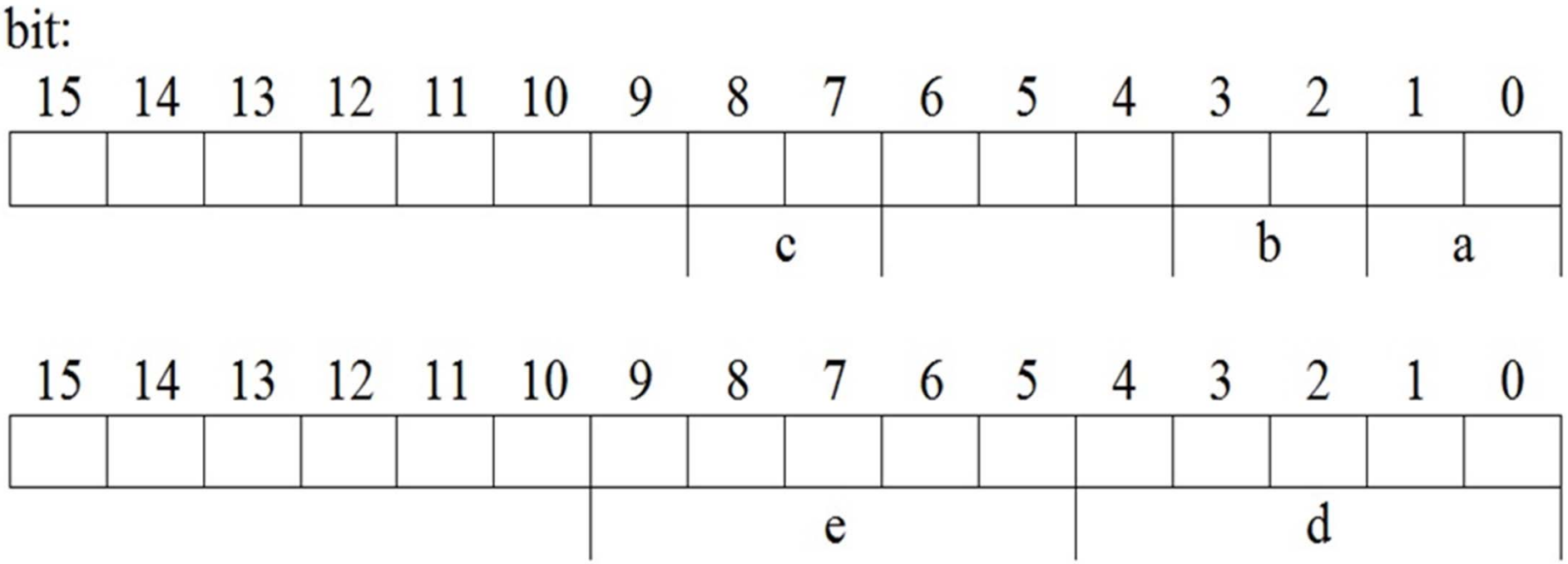
- As a special exception to the usual C syntax rules, it is implementation-defined whether a bit field declared as type **int**, without specifying signed or unsigned, is signed or unsigned. Thus, it is recommended to explicitly specify signed or unsigned on all structure members for portability.

- Unnamed fields consisting of just a colon followed by a number of bits are also allowed; these indicate **padding**. Specifying a width of zero for an unnamed field is used to force alignment to a new word.

Example:

```
struct
        { unsigned a:2;
          int b:2;
        unsigned :3;
        unsigned c:2;
        unsigned :0;
        int d:5;
        unsigned e:5;
        }x,y;
```

For x, two words are allocated in the following way:



Notes:

- The members of bit fields do not have addresses, and as such cannot be used with the address-of (***&***) unary operator.

- The ***sizeof*** operator may not be applied to bit fields.

7. Enumerated type

- In C, enumerations are created by explicit definitions, which use the **enum** keyword and are reminiscent of *struct* and *union* definitions.
- Enumerated type allows the programmer to use meaningful names for numeric values.
- For example, the name of the month of the year, January can be associated with the value of 1, February associated with value 2, etc.
- Enumeration type is used when it is not desired to see successive integers, but the some symbols associated with these numbers.

The general format of enumeration type is :

enum name{nume_0,nume_1,nume_2,...,nume_k} v1,v2,...,vn;

where:

- ***name*** is the name of the enumeration type introduced by this statement;
- ***nume_0, ..., nume_k*** are names that will be used instead of numerical values (***nume_i*** has value ***i***);
- ***v1, v2, ..., vn*** are data, which are stated with the ***name*** type (are similar to int type).

As observation, data can be later defined as ***name*** type:

enum name v1,v2,...,vn;

Examples:

1)

enum {ilegal, jan, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec} month;

In this case, the assignment

month = 4;

is equivalent with the more suggestive format

month = apr;

and *month == 8*

is equivalent with

month == aug

2) *enum Boolean {false,true} a;*

The expression

a==false

is equivalent with

a==0

and

a==true

is equivalent with

a==1

3) *enum S{s1, s2, s3=100, s4, s5};*

Associated values are: s1=0; s2=1; s3=100; s4=101; s5=102.

8. Recursively defined data

- Recursively defined data refers to data structures that are declared in a special format, which contains a form of recursivity.

- More specifically, a recursive data type is a data type for values that may contain other values of the same type (**which refers to itself**).

- To clarify things, we can consider the structure:

```
struct name
        {
         statements;
        };
```

Notes:

- The statements should include various data types (predefined or user) but different from **name** type.

- But the statements could define pointers of the **name** type (in the case of **recursively defined data**).

Consequently, the following statement

```
struct name
        {statements;
        struct name *name1;
        statements;
        };
```

is correct, while:

```
struct name
        {statements;
        struct name name1;
        statements;
        };
```

is incorrect.

Definition

If a user type has at least one component that is a pointer to itself, then it is a recursive type.

Example:

```
typedef struct node
                { char *word;
                  int frequency;
                  struct node *next;
                } NOD;
```

Notes :

- An important application of recursion in computer science is in defining dynamic data structures such as Lists and Trees.
- Recursive data structures can dynamically grow to a theoretically infinite size in response to runtime requirements.
- In contrast, a static array's size requirements must be set at compile time.

Cap. Lists
1. Introduction

- Linked lists are the best and simplest example of a dynamic data structure that uses pointers for theirs implementation.
- Understanding pointers is crucial to understanding how linked lists work.
- Essentially, linked lists function as an array that can grow and shrink as needed, from any point in the array.

Definition

- **As a first definition, a list is a dynamic group, meaning that it has a variable number of elements.**

- At the beginning, a list is an empty group. During the program execution, we can add new items to the list and also various elements can be removed from the list.

- Essentially, linked lists function as an array that can grow and shrink as needed, from any point in the array.

- Advantages of dynamic lists over arrays:
  - Items can be added or removed from ***any position on the list***
  - There is no need to define an initial size

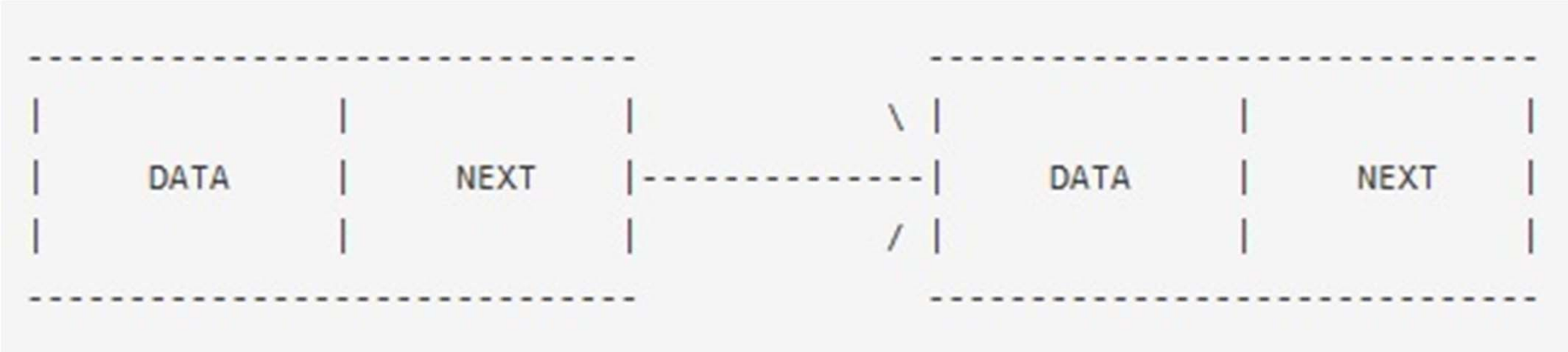
But linked lists also have a few disadvantages:

- There is no "random" access - it is impossible to reach the ***n*-**th item in the array without first iterating over all items up until that item. This means we have to start from the beginning of the list and count how many times we advance in the list until we get to the desired item.
- Dynamic memory allocation and pointers are required, which complicates the code and increases the risk of memory leaks and segment faults.
- Linked lists need a much larger space over arrays, since linked list items are dynamically allocated (which is less efficient in memory usage) and each item in the list also must store an additional pointer.

What is a linked list?

- A linked list is a set of dynamically allocated nodes, arranged in such a way that each node contains one value and (at least) one pointer. The pointer always indicates to the next member of the list. If the pointer is NULL, then there is the last node in the list.

- A linked list is held using a local pointer variable which points to the first item of the list. If that pointer is also NULL, then the list is considered to be empty.


DATA
NEXT
DATA
NEXT

• The ordering of the elements of a list is done using pointers to the list elements (each such pointer is a part of the node structure).

• Due to these pointers, list items are recursive structures.

• Lists organized in this way are called linked list.

Another definition (more comprehensive):

A group of dynamic recursive structures of the same type, which includes one or more relationships for linking elements (using pointers from these structures), is called a **linked list**.

- The elements of a list are called **nods.** If between the nods of a lists is only one order relation, then the list is called **simple linked**.

- Similarly, the list is **double linked** if in between the nods are defined two order relations.

- A list is **n-chained** if between every two successive nodes there are defined ***n*** relations for order .

Operations related to a linked list:

a) creation of a linked list;

b) access to any node of the list;

c) inserting a node in a linked list;

d) deleting a node from a linked list;

e) deleting a linked list.

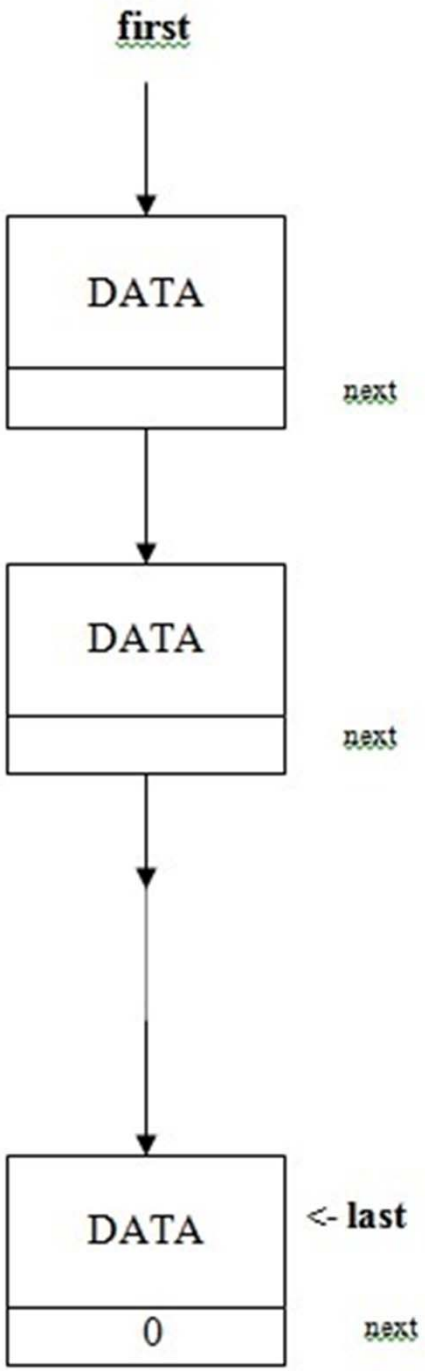
2. The simple linked list

Between the nodes of the simple linked list we have only one order relation. There is a single node that doesn't have a successor and a single node that doesn't have a predecessor.

These nodes are the edges of the list.

We will use two pointers to the front and back edges of the list, which we'll denote with:

- **first** – the pointer to the node that has no predecessor;
- **last** – the pointer to the node that has no successor.



The **next** pointer defines the successor relation for the list's nodes. This is a pointer for connexion. At each node, it has as value the address of the next node from the list. An exception is the linked pointer from the last node (for which **next** takes the value zero).

2.1. Creating a simple linked list

When a simple linked list is built, we are doing the following tasks:

a) initialize both pointers **first** and **last** with zero value, because at the beginning the list is empty
b) allocate the memory zone in the heap memory for the current node **p**
c) load the current node **p** with the correspondent data (if it exists), and then go to the next step **d)**. Otherwise eliminates p and returns from the function.

d) assign the addresses of the list, for the current node:
last -> next = p, if the list is not empty;
first = last = p, if the list is empty.

e) the current node is assigned with the pointer that denotes the last element of the list (**last=p**)

f) **last -> next** = 0

g) the process jumps to point **b)** in order to add a new node to the list.

Notes:

- It is considered the following type:

```
typedef struct node
        { <statements>;
          struct node *next;
        } NOD;
```

- A special function (called **incnod**) is used, which loads the current data in a node of NOD type.

As an example, let's define a linked list node:

```
typedef struct node
{       int val;
        struct node * next;
} NOD;
```

- The **incnod** function returns:
  - **1** to mark correct loading of data in the current node
  - **-1** when data are not loaded
  - **0** when an error occurs (eg insufficient memory)
- The prototype of **incnod** function is:

```
int incnod(NOD *p);
```

- Another required function releases the memory area reserved for a specified node:

```
void elibnod(NOD *p);
```

2.2. The access to a node in a simple linked list

- We can have access to the nodes of a simple linked list, starting with the **first** node, and then by passing from a node to another one, using the **next** pointer.

- A better method is to have a **key** date, (a component of the nodes) that should have different values, for each node. In this case it can be defined an access to the lists' node based on the value of this **key** (usually with the **char** or **int** type). The function returns the pointer to the identified node or zero, if there is no match to the **key**.