C Programming, Autumn 2013, Exercises for the Second Week

Notice: Remember that you can find information about a standard C library function by writing `man 3 function_name` in the terminal, or by going to the address:

`http://linux.die.net/man/3/function_name`

In this week exercises we learn to handle arrays and pointers. As a prerequisite for a successful programming you should know the various list data structures and their implementation principles.

1. Write a function

   ```
   void save_sum(int* x, int* y, int* z)
   ```

   that stores the sum of the integers pointed by x and y into the memory word pointed by z.

2. Write a function

   ```
   double average(double* a, int len)
   ```

   that returns the average value of len decimal numbers. The numbers are stored sequentally in the memory starting from the word pointed by a.

3. Write a function

   ```
   int substitute_and_sum(int* a, int len)
   ```

   that first stores 2 into two bytes starting from $\lfloor len/2 \rfloor$ in the array a, and then calculates the sum of the short integers in the array in the following way: The array is intepreted as an array of short integers (two bytes). These integers are summed. The function returns the sum.

   The following optional exercise may help to understand the task.

   Assume that the following expressions are true:

   ```
   sizeof(short) == 2
   sizeof(int) == 4
   sizeof(float) == 4
   sizeof(double) == 8
   ```

   Also, assume that we have declared the array

   ```
   int arr[5] = { 0, 0, 0, 0, 0 };
   ```

Explain with a picture (hand-drawn is fine) where the following statements store values (both the location and the amount of bytes written). Assume that all the assignments succeed.

a) `arr[3] = 42;`

b) `arr[9] = 7;`

c) `arr[-4] = 1;`

d) `((short*)arr)[7] = 128;`

e) `((double*)arr)[2] = 3.14;`

f) `((char*)&arr[1])[6] = 'A';`

g) `((float*)(&((short*)&arr[3])[-3]))[0] = 7.5;` (This is challenging!)

(Hint: Remember that array indexing [] binds more tightly than the address-of operator & and type conversions. Start analysing the more complex expressions from the inside.)

If all the previous assignments succeed, which of the assignments have affected the value of:

i) `arr[0]`

ii) `arr[2]`

iii) `arr[4]`

4. Write a function

```
void shuffle_ints(int* array, int len);
```

that shuffles the contents of the given array of integers. The parameter len contains the number of int values in the array. Write the shuffling code so that, given a random function with uniform distribution, every possible ordering of the contents is as likely. (Use the rand function provided by the standard C library.)

Example call:

```
int nums[] = { 0, 1, 2, 3, 4, 5, 6 };
shuffle_ints(nums, 7);
```

Example result:

```
1, 0, 6, 4, 3, 2, 5
```

5. In the exercise 1 we implemented operations for fractions. Represent now a fraction with the help of a structure:

```
typedef struct {
     int numerator;
     int denominator;
} fraction;
```

This structure is found in the included header file. Implement the addition operation so that the function returns the sum in a structure:

```
fraction add_fractions(fraction r, fraction s);
```

6. Consider a doubly linked list which contains integer values (see the figures in the appendix). Its type is found in the included header file and is defined as follows:

```
typedef struct node{
  int val;
  struct node * prev;
  struct node * next;
} dList;
```

Write a function

```
dList* insert_element_d(dList *L, dList *p, int value);
```

which adds an integer *value* into the list L to the right side of position p. Take into account in this and other exercises that initially the list may be empty. If p is NULL, then insert the element in the beginning of the list. The function returns a pointer to the start of the list. The operation should work in a constant time.

7. Write a function

```
int print_elements_d(dList *L);
```

which prints the integers in the list one per line so that the values are printed followed by a line break. Notice that printing the list should not have side effects. If L is null return 1, else return 0.

8. Write a function

```
dList* delete_element_d(dList *L, dList *p);
```

which deletes the element pointed by p. The function returns a pointer to the start of the list. The function should work in a constant time.

9. Write a function

```
int order_list_d(dList *L);
```

which orders the values in the doubly linked list L from the smallest to the largest. The function returns 0, if everything succeeds, otherwise 1.

10. Write a function

```
dList*  merge_lists_d(dList *L1, dList *L2);
```

which merges ordered doubly linked lists Li and L2 into a third list which is ordered, too. The function returns the third list and it should work in a linear time. The original lists are left intact. When a value is found in both lists, include it only once.

11. Consider a singly linked list which contains integer values. The lists have a header node. The types are defined as follows in the included header file:

```
typedef struct snode{
    int value;
    struct snode * next;
}snodeType;

typedef struct hnode{
    int count;
    snodeType* first;
    snodeType* last;
} sList;
```

Write the function

```
sList* create_sList(void);
```

which creates an empty list.

12. Write a function

```
int insert_element_s(sList *L, sNodeType *p, int value);
```

which adds an integer *value* into the list L to the right side of position p. If p is NULL, then insert in the front of the list. The function returns 0, if everything succeeds, otherwise 1.

13. Write a function

```
int  delete_element_s(sList *L, sNodeType *p);
```

which deletes an element on the right side of the element pointed by p. If p is NULL, then the first element is deleted. The operation should work in a constant time. The function returns 0, if the operation succeeds, otherwise 1. Deleting a NULL element is considered unsucessful.

14. Write a function

```
sList*  merge_lists(sList *L1, sList *L2);
```

which merges ordered singly linked lists L1 and L2 into a third singly linked list which is ordered, too. The function should work in a linear time and return the third list. The original lists are left intact. When a value is found in both lists, include it only once.

APPENDIX: List Diagrams

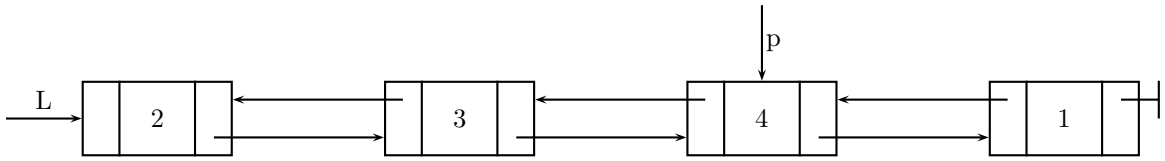

Figure 1: Doubly linked list

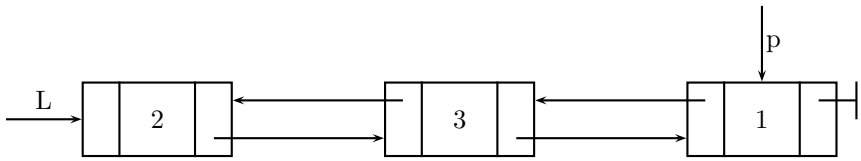

Figure 2: After deleting the item 4 pointed by p

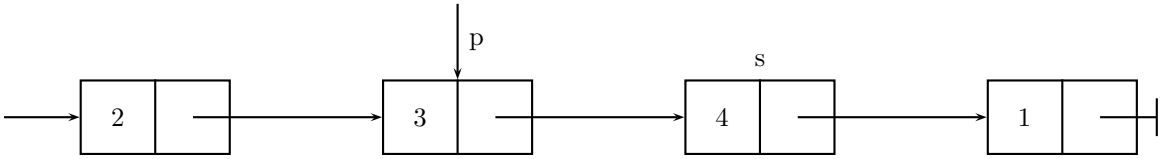

Figure 3: Singly linked list and position s pointed by p

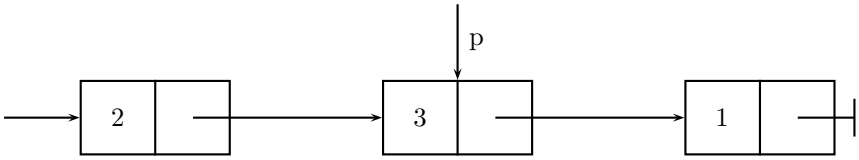

Figure 4: Singly linked list after deleting the node s