Informatics II, Spring 2024, Solution Exercise 6

Publication of exercise: April 8, 2024
Publication of solution: April 15, 2024
Exercise classes: April 15 - April 19, 2024

Learning Goal

- Deepen the understanding of Pointers and Linked List.
- Being able to apply them to further advanced ADTs.

Task 1 [Easy]

Complete the following table. *Hint: Draw or try it out if it helps.*

C code	**What is it?**
`int i;`	*Integer somewhere in memory*
`int *p;`	*Pointer to an integer, contains address of the integer (e.g. `=i`)*
`&i`	*Pointer to address of the integer `i`*
`p`	*Address of an integer*
`*p`	*Follow pointer to get what's at the address `p`*
`&p`	*Address of pointer to an integer*
`int **q;`	Pointer to a pointer to an integer (same as `&p`) (e.g. `=p`)
`*q`	*Pointer to an integer (Follow pointer to get to pointer to an integer)*
`**q`	*Integer (Follow pointer to pointer to integer)*

(Bonus) Pointer Arithmetic

Let `a` be an integer array with length of 10.

C code	**What is it?**
`int a[10];`	*Integer array `a` of size 10*
`a[0]`	*Reference to the 0-th element in `a`*
`int *p = &a[0]`	*Sets `p` to point to the 0-th elment in `a`*
`*p`	*Content of `a[0]`*
`int *pa = a;`	Pointer to the first element in the array (same as `&a[0]`)
`*pa`	*Content of `a[0]`*
`*(pa+1)`	*Follow pointer to the second element and get the value (same as `a[0+1]`)*
`*pa+1`	*Follow pointer to first element and add one (same as `a[0]+1`)*

We can learn that arrays are basically pointers. `a` is just a pointer to the first elment of the array. This is different as in Python, where the array is an object. So when `int *p = a` is the same thing as `int *p = &a[0]`. This is the foundation of pointer arithmetics.

As a bonus of the bonus: It does not matter what datatype the array holds. The concept of pointer arithmetics automatically progresses to the next element on `(p+i)` with the corresponding number of bits.

Task 2 [Medium]

Given is the following struct of a node. Each node represents a month, with some properties, and is in a linked list.

```c
struct month {
    int month_number;
    char *month_name;
    struct month *next;
};
```

After compiling the script, a linked list of these month nodes is initialized in a random order. (You do not need to understand how they are initialized for this exercise.) The linked list is stored in the `head` pointer, which points to the head of this month linked list. Please note that this is a single linked list, with only a pointer to the head (no tail pointer).

Use the template `C` code in the `./task2.c` file. You do not have to modify the `init_months()` and `free_months()` function. These only serve as setup and teardown of the linked list. Your tasks are listed below.

(a) Print Linked List

As a first task, implement the `print_months()` function. This function should print each month node in the linked list. Use the following format for the print statements: `"January (1)"`.

```c
void print_months(struct month *head) {
    printf("months: ");
}
```

(b) Get Previous Node

Sometimes we want to know the preceding month of a given month in our linked list. Implement the `get_previous_month()`, which returns a pointer to the preceding month. *Hint: You might want to add some parameters.*

```c
struct month * get_previous_month() {
    return NULL;
}
```

Hint: Test your implementation.

(c) Swapping Nodes

It is useful to be able to swap two months in the linked list. Implement the `swap_month(struct month *head)` function, which returns a pointer to the (new) head. The swapping should swap the entire node, **not** the node values.

```c
struct month* swap_month(struct month *head, struct month *a, struct month *b) {
    return head;
}
```

Think about differet cases which might occur. You can assume, that `a` comes always before `b` in the linked list.

(d) Selection Sort

Now we want to combine the previously implemented functions to create a **in order** selection sort algorithm. For that implement the `selection_sort()` function, which returns a pointer to the (changed) head. Keep in mind that this implementation is in order, which means that you should not create a second head or linked list.

```
1 struct month *selection_sort(struct month *head) {
2     return head;
3 }
```

Hint: Use the previously implemented functions and print for every step the current linked list with `print_months()`.

Solution(a),(b),(c),(d): see code month_sort.c

Task 3 [Medium]

(a) Merge

Consider two sorted doubly linked lists. Implement a function `merge()`, which merges those two lists into a single doubly linked list. The resulting, merged, list should be sorted and containing all elements from the input linked lists. The `merge()` function should not copy values of the input linked lists. Use the following `struct` for a node.

```
1 struct node {
2     int val;
3     struct node *next;
4     struct node *prev;
5 };
```

Solution: see code merge.c

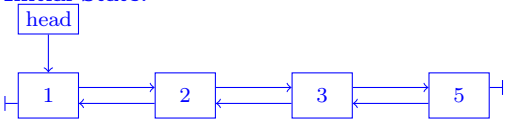
(b) Tenet

Let the same `struct` as in Task 3(a) define a node in a **doubly linked list**. Implement a function `reverse()` which takes a pointer to a node and returns the pointer to the head of the reversed list. The `reverse()` function should only change pointers. It should not create new nodes.

There are two ways to implement this. The easier way is to implement an iterative algorithm. A harder, but more elegant solution is to implement an recursive function.

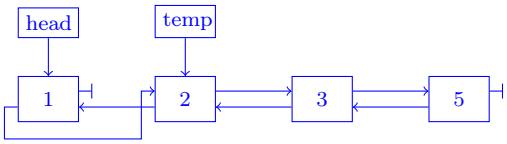
Solution: see code reverse.c
Recursive Solution:

Initial State:



Swapping on Node 1:

- Swap the pointers of `head = 1`
  1. `temp = head->next = 2`
  2. `head->next = head->prev = NULL`
  3. `head->prev = temp = 2`



- Analog with next `head->prev` in this case with node `2`, etc.