05OGDLP - Algorithms & Programming
Laboratory 7

Learning objectives

- Sorting
- Dynamic memory allocation
- Using `void *` and function pointers to write generic code
- Abstract Data Types (ADT)
- Double-linked lists

1 Task Manager

Implement a task manager backed by a double-linked list ADT. The program will be menu-driven, offering the following functionalities:

- Add a task
- Find a task by ID and display it
- Remove a task by ID and display it
- Display tasks in sorted priority order (highest to lowest)
- Modify the priority of an existing task

A `task` is characterized by the following fields

- `id`: a unique identifier for the task
- `priority`: priority of the task, with values ranging from 139 (lowest priority) to 0 (highest priority)
- `description`: a brief description of the task
- `duration`: estimated duration of the task (in seconds)

The `tasks` are stored in a (sorted) double-linked list ADT, i.e., a generic *container*.

1.1 Generic programming

In C, generic programming is a way to write code that can operate on different data types. This is achieved using `void *`, a pointer to any data type. Using `void *` allows the list ADT to store any data type, making it more versatile and reusable. In your case, the list needs to store `task` objects. By using `void *` for data elements, the same list ADT can handle `task` objects, integers, or any other data type.

1.2 Implementing the list ADT

An ADT is a data type defined by its operations rather than its implementation. It provides a logical description of how the data type behaves. The benefits it provides are encapsulation and abstraction of data, allowing for reusable code.

Our double-linked list ADT is defined by the following operations (in `list.h`):

- `list_t * list_new()`: allocates and returns a new list
- `void list_free(list_t *, void (* data_free)(void *))`: frees the list and its elements
- `int list_size(list_t *)`: returns the number of elements in the list
- `int list_insert(list_t *, void *, int (* data_compare)(void *, void *))`: inserts a new element into the list. If `* data_compare` is not `NULL`, it ensures that the list remains sorted after each insertion. To maintain the abstract nature of the `list` ADT and allow for sorting based on various criteria, we pass the `*data_compare` function as a function pointer to `list_insert`. This approach will enable the list to be sorted by any criteria defined outside the list implementation, adhering to the principles of abstract data types. `list_insert` will iterate through the list, using the `data_compare` function to find the correct insertion point for the new element. The client code (i.e., where the list ADT is used), will define the `data_compare` function, and `list_insert` will be called with `data_compare` as an argument (or NULL, to append to the front/tail of the list).
- `void * list_remove(list_t *, void *, int (* data_compare)(void *, void *))`: removes an element from the list based on a comparison function, and returns the removed data
- `void * list_search(list_t *, void *, int (* data_compare)(void *, void *))`: finds an element in the list using a comparison function. Returns the data if a match is found, `NULL` otherwise.
- `void list_print(list_t *, FILE *, void (* data_print)(void *, FILE *))`: prints the list contents.

Carefully read through the following example where `person_t` is used as a list item, to understand how the `list` ADT behaves:

```
#include "list.h"

typedef struct{
    char name[100];
    char surname[100];
    int age;
}person_t;

person_t * person_new(){
    person_t * p = (person_t *)malloc(...);
    ...
    return p;
}

void person_free(void * p){
    free(p);
}

int person_compare_age(void * elem_a, void * elem_b){
    // cast the void * arguments to person_t *
    person_t * p1 = (person_t *)elem_a;
    person_t * p2 = (person_t *)elem_b;
    // compare the age field of the two person_t items
    return p2->age - p1->age  // sorts from older to younger
}

int person_compare_name(void * elem, void * key){
    ...
}

void person_print(void * elem, FILE * fp){
    person_t * p = (person_t *)elem;
    fprintf(fp, "%s %s %d\n", p->name, p->surname, p->age);
}

int main(){
    list_t * l = list_new(); // create new empty list
    // create p1, p2, p3
    person_t * p1 = person_new();
    person_t * p2 = person_new();
    person_t * p3 = person_new();
    // initialize p1, p2, p3 with some data...
    list_insert(l, p1, NULL); // non-sorted insertion
    list_insert(l, p2, person_compare_age); // sorted insertion
    list_insert(l, p3, person_compare_age);
    // find person by name and print them
    char key[] = "John";
    person_t * result = list_search(l, (void *)key, person_compare_name);
    person_print(result, stdout);
    list_print(l, stdout, person_print); // print the whole list
    list_free(l, person_free); // release all memory
    return 0;
}
```

1.3 Internal mechanics of the list

Internally, our `list_t` is implemented as a structure having pointers to the first and last `node_t` elements, which represent the head and tail of the list.

The `node_t` structure serves as the fundamental building block of the list. Each `node_t` represents an individual element, containing a `void *` pointer to hold data of any type, thus making the list generic. In addition to the data, each `node_t` has pointers to the next and previous nodes, allowing for efficient traversal in both directions.

```
typedef struct node {
    void *data;          // pointer to the generic data
    struct node *next;   // pointer to the next node
    struct node *prev;   // pointer to the previous node
} node_t;

typedef struct list {
    node_t * head;      // pointer to the first node in the list
    node_t * tail;      // pointer to the last node in the list
    // other metadata like size of the list can be added here
} list_t;
```

1.3.1 Adding elements

When adding an element to the list, a new `node_t` is dynamically allocated. The `data` pointer in `node_t` is set to point to the element being added. The new node is then linked into the list by adjusting the `next` and `prev` pointers to the adjacent nodes.

1.3.2 Removing elements

To remove an element, the list is traversed to find the `node_t` containing the element. Once found, the `prev` and `next` pointers of the adjacent nodes are adjusted to bypass the node being removed. Finally, the `node_t` is deallocated, ensuring no memory leaks.

1.4 Additional considerations

- Ensure that all list manipulation functions (like `list_remove` and `list_search`) are compatible with the sorted nature of the list.
- Consider edge cases, such as inserting a task when the list is empty, inserting at the beginning or end of the list, and dealing with tasks having the same priority.

1.5 Extension: task execution simulation

Add a new feature in the task manager for simulating task execution. The simulation should process tasks based on their priority, display relevant task data and timings, and then remove the task from the list. Steps to implement:

- Update the main menu to include an option for task execution simulation
- Use a mechanism to keep track of UTC times. You can use time-related functions in C, like `time()`, to get the current UTC time
- Simulate non-preemptive task execution. Start processing tasks from the list, beginning with the highest priority task (lowest numerical value). For each task, print starting time, task data, and calculated ending time (use the `duration` field in `task_t`). Remove the task from the list.

1.5.1 Example

```
Task Execution Simulation Starting...

1. Executing Task
   Starting UTC Time: 2023-11-21 08:00:00
   Task Data: [ID: 201, Description: "System Update", Priority: 1]
   Ending UTC Time: 2023-11-21 08:03:11
   Task completed and removed from the list.

2. Executing Task
   Starting UTC Time: 2023-11-21 08:03:11
   Task Data: [ID: 202, Description: "Disk Cleanup", Priority: 2]
   Ending UTC Time: 2023-11-21 09:25:03
   Task completed and removed from the list.

3. Executing Task
   Starting UTC Time: 2023-11-21 09:25:03
   Task Data: [ID: 203, Description: "Security Scan", Priority: 3]
   Ending UTC Time: 2023-11-21 09:35:00
   Task completed and removed from the list.

4. Executing Task
   Starting UTC Time: 2023-11-21 09:35:00
   Task Data: [ID: 204, Description: "Network Diagnostics", Priority: 4]
   Ending UTC Time: 2023-11-21 09:36:21
   Task completed and removed from the list.

5. Executing Task
   Starting UTC Time: 2023-11-21 09:36:21
   Task Data: [ID: 205, Description: "Log Archiving", Priority: 5]
   Ending UTC Time: 2023-11-21 10:00:00
   Task completed and removed from the list.

Simulation Completed. All tasks executed and removed from the list.
```

2 Conway's Game of Life (part 3)

This is the last part of our implementation of Conway's Game of Life. We are going to extend the program so that the game runs *forever*, and meaningful initial states can be read from a file. The set of milestones you will have reached by the end of this part is:

1. Building a data structure to store the board state
2. Initializing the board to a random state
3. Printing the board on standard output
4. Calculating the next state of the board
5. Running the Game of Life forever
6. Loading initial states from file

5. Running the Game of Life forever

Modify the `main` function provided in *Testing the library* (in part 1, Exercise 2 Laboratory 3) so that the next state is computed and the updated grid is shown until there is at least one cell alive. To do so, write an additional function

```
int all_dead(grid_t *);
```

that returns `0` when the number of alive cells is greater than zero, and `1` otherwise. To print your updated grid on top of the old one, update the `grid_show` function you implemented in part 1 by adding the following as its first line (an explanation is here):

```
printf("\e[1;1H\e[2J");
```

6. Loading initial states from file

So far, you have initialized Life from a random state, a so-called "soup". Soups usually evolve for a few generations, and gradually fade out. However, many different types of patterns occur in the Game of Life, which are classified according to their behavior. Common pattern types include still lifes, which do not change from one generation to the next; oscillators, which return to their initial state after a finite number of generations; and spaceships, which translate themselves across the grid. You can see some examples on this page.

Write a function

```
void grid_state_from_file(grid_t *, char *);
```

that expects a grid and an input file name as parameters and returns nothing. The function initializes a `grid_t *` that has previously been allocated (using the function `grid_new` you implemeneted in part 1). The input file is a binary

matrix. The first line indicates the number of rows and columns of the input state. The second line indicates the offset (from the grid `(0, 0)` origin) to copy the input state to, i.e. a `3 4` offset means that the input state must be copied starting at the 3rd row and 4th column in the grid. If the binary matrix (plus the offset) is smaller than the grid, assume all zeros for the rest. If larger, copy only the rows and columns of the input state that can fit into the grid.

Test your function on the following basic patterns – a *toad* (Listing 1, Fig. 1) and a *glider* (Listing 2, Fig. 2) – an oscillator and a spaceship, respectively:

Listing 1: Binary matrix for the "toad" pattern, to be placed at offset (3, 3)

```
4 6
3 3
000000
001110
011100
000000
```

Listing 2: Binary matrix for the "glider" pattern, to be placed at offset (10, 4)

```
3 9
10 4
000010000
000001000
000111000
```

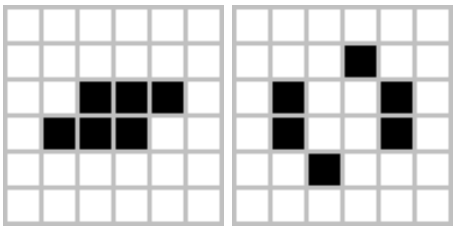

Figure 1: Evolution of a "toad" pattern

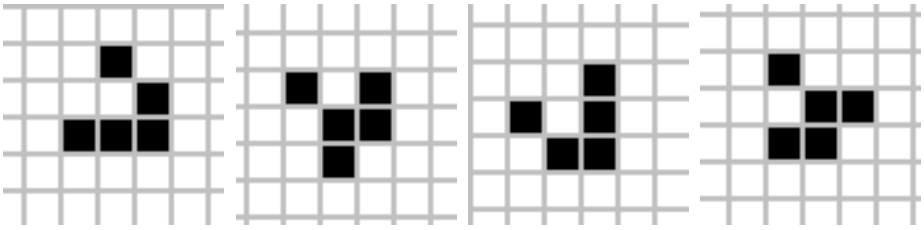

Figure 2: Evolution of a "glider" pattern