TP 4

1 Heaps and priority queues

The goal of this exercise is to implement a priority queue data structure. Our implementation will be based on the notion of heap, which we have studied in class, with the additional feature that priority values should be updatable[1]. Such feature is absent from (e.g.) the C++ standard library.

As a warm-up we shall start by implementing a standard heap based priority queue.
The elements in the heap array will be of the following form, where `id` is their unique identifier and `val` is their priority value.

```
struct priority_data {
    int id;
    float val;
};
```

The priority queue itself is nothing more than dynamical array of `struct priority_data` implementing a heap:

```
struct priority_queue {
    struct priority_data *heap;
    int capacity;
    int size;
};
```

Exercise 1. Implement the following functions:

```
int priority_queue_init(struct priority_queue *q);
void priority_queue_push(struct priority_queue *q, int id, float val);
struct priority_data priority_queue_pop(struct priority_queue *q);
```

The last two functions will make use of the following elementary heap operations, which should not be part of the API (hence static functions in the .c source file):

```
static void swap(struct priority_queue *q, int pos1, int pos2);
static void sift_up(struct priority_queue *q, int pos);
static void sift_down(struct priority_queue *q, int pos);
```

Recall that we have defined sift-up and sift-down in class as the operations which make a node in the heap be recursively swapped with its parent (for sift-up) or with one of its children (for sift-down) until the heap property is satisfied. Wether you implement these recursively or not is up to you, but non recursive versions are potentially faster if implemented properly.

Exercise 2. Our next goal is to implement a function:

```
void priority_queue_update(struct priority_queue *q, int key, float new_val);
```

Since we have no cost effective way yet to search for `key` in our priority queue heap[2], we shall modify slightly our structure. One possibility would be to add a dictionary `key` $\to$ `pos_in_heap`, e.g. through a hash table or a binary search tree, and update it at each push or pop. Here we shall deal with the simpler situation where `id`s are assumed sufficiently small so that they can be used directly as indices into a `pos_in_heap` array (i.e. the position in `heap` of the data with id `id` is stored in `pos_in_heap[id]`). Such a situation is common when the id is already an index into an existing user data array.

[1]This is a desirable feature in some algorithms, in particular the famous Dijkstra algorithm which will learn in class.
[2]Basically our only strategy would be linear search in the heap array.

```
struct priority_queue {
    struct priority_data *heap;
    int capacity;
    int size;
    int *pos_in_heap;
};
```

The initialization function should be modified into

```
int priority_queue_init(struct priority_queue *q, int max_id);
```

and the latter will also allocate an array of `max_id + 1` integers and pre-fill it with -1 (which we will understand as the code for the corresponding id being not present yet in the priority queue, making the assumption that effective ids are all non negative).
The `pos_in_heap` array should be updated at each push or pop operations. This will imply some modifications in the code for the `swap`, `priority_queue_push` and `priority_queue_pop` functions.

2 (Intrusive) AVL trees

In class we have studied the notion of binary search tree, including the version called AVL which had an auto-balancing property using tree rotations as needed. In this exercise you will implement AVL trees for arbitrary user data equipped with an ordering. We use a so-called *intrusive* data structure, meaning the tree nodes will note contain user data as a struct field, but instead the user data structs will need to define our tree nodes as part of themselves. In particular, memory allocation is entirely delegated to user code, and we shall not deal with it.

More precisely, we define the tree node structure:

```
struct avl_node {
    struct avl_node* left;
    struct avl_node* right;
    int height;
};
```

and the tree structure:

```
struct avl_tree {
    struct avl_node* root;
    int (*cmp)(const void *, const void *);
    int offset;
};
```

The user structure (assuming it is called `user_data` here), will be of the form

```
struct user_data {
    /*
     * whatever struct fields before node, possibly none
     */
    struct avl_node* node;
    /*
     * whatever struct fields after node, possibly none
     */
};
```

The `offset` field of the `struct avl_tree` data structure needs to be initialized with the offset of the field `node` in `struct user_data`. In C/C++ this can be obtained by the function `offsetof(struct user_data, node)` (the general prototype is `offsetof(structname, fieldname)`). This offset allows to obtain the address of the user data from the address of the node (simply by subtracting offset from the latter).

The cmp field of the struct avl_tree data structure also needs to be filled at initialization, it is the address of the function responsible for comparing two struct user_data. In C/C++, the address of a function is given simply by its name.
The prototype for our struct avl_tree structure will thus be

```
void avl_tree_initialize(struct avl_tree *t,
                         int (*cmp)(const void *, const void *),
                         int offset);
```

Exercise 1. In addition to the initialization function above, implement the following API functions:

```
void *avl_tree_find(struct avl_tree *t, const void *data);
void avl_tree_insert(struct avl_tree *t, void *data);
```

These will make use of non API functions, in particular rotations.
Then test your implementation in a main.c file with the user data of your choice (e.g. strings).

Exercise 2. (More challenging) A less trivial but more efficient implementation can be based on the following

```
struct avl_node {
    struct avl_node* children[2];
    uintptr_t pcb;
};
```

The replacement of left and right child pointers by an array of two can be used to avoid some branches in your code and symmetrize it.
More importantly, the pcb field (acronym for parent-child-balance) packs three informations into one (the type uintptr_t is an unsigned integer type sufficiently large to contain any memory address of the system). The lower two bits are used to store the balance factor of the node (shifted by one so that it fits into $\{0, 1, 2\}$). The third lower bit indicates whether the node is the left child or right child of its parent. Finally the remaining bits correspond to the address of the parent node (or more precisely: the address of the parent node is obtained from pcb by clearing the lowest three bits). This works provided tree nodes are always 8bytes aligned in memory (meaning that their address is a multiple of 8), which will always be the case on 64bit systems.

You will try to avoid recursion as much as possible in your implementation (it can be completely eliminated). Having the parent pointer available helps in speeding-up other common tree API functions which we have not touched here, in particular tree traversal, which you can implement as well.