# EXERCISES — Dlist

version #

ASSISTANTS C/UNIX 2022 <assistants@tickets.assistants.epita.fr>

## Copyright

## Contents

*https://intra.assistants.epita.fr

# 1 Dlist

**Files to submit**:

- dlist/Makefile
- dlist/*

**Provided files**:

- dlist/dlist.h

**Makefile:** Your makefile should define at least the following targets:

- library: Produce the libdlist.a library
- clean: Delete everything produced by make

**Authorized functions:** You are only allowed to use the following functions:

- malloc(3)
- calloc(3)
- free(3)
- putchar(3)
- puts(3)
- printf(3)

**Authorized headers:** You are only allowed to use the functions defined in the following headers:

- err.h
- errno.h
- assert.h
- stddef.h

## 1.1 Goal

In this exercise you must implement a doubly linked list.

A doubly link list is similar to a normal linked list, but each element contains a reference to the previous element. Furthermore, on top of the reference to the head, a doubly linked list also stores a reference to its tail.

The structures that you will use are:

```c
struct dlist
{
    size_t size;
    struct dlist_item *head;
    struct dlist_item *tail;
};
```

(continues on next page)

(continued from previous page)

```c
struct dlist_item
{
    int data;
    struct dlist_item *next;
    struct dlist_item *prev;
};
```

You are free to name your files however you like, as long as the coding style is respected.

## 1.2 Preamble

- The header file `dlist.h` is provided on the intranet. It is mandatory to use it as is, in order to respect functions prototypes and data structures. You may add other header files though, if the `make library` command still generates your library.
- Pay attention to your insertion and creation functions. These are essential to the tests.
- Manage your memory wisely, the exercise will be graded checking memory leaks.
- The case of a `NULL` pointer to your `dlist` structure will not be tested.

  > You still have to handle `malloc` errors. In case of an error, your function must not behave as expected (example: no modification). If the function should have returned a pointer, `NULL` must be returned instead.

- Lists can only contain **positive** integers: a negative integer must raise an error.
- The `size` field of the structure must always be up to date.
- You must find a way to optimize your index accesses to the list.
- We consider that a list is sorted when the elements are in ascending order: the smallest element being the head.
- You are encouraged to add your own functions. To do so you are free to add source files.

## 1.3 Threshold 1

### 1.3.1 dlist_init

```c
struct dlist *dlist_init(void);
```

Return a new allocated empty list.

### 1.3.2 dlist_push_front

```c
int dlist_push_front(struct dlist *list, int element);
```

Add `element` to the front of `list`. In case of failure return `0`. Otherwise, return `1`.

### 1.3.3 dlist_print

```c
void dlist_print(const struct dlist *list);
```

Display the elements of the list from head to tail. A line feed must delimit each element. A line feed must also be printed after the last element. If the list is empty, nothing must be printed.

Example:

```c
int main(void)
{
    struct dlist *l = dlist_init();

    dlist_push_front(l, 3);
    dlist_push_front(l, 2);
    dlist_push_front(l, 1);

    dlist_print(l);

    return 0;
}
```

Should print:

```
42sh ./a.out | cat -e
1$
2$
3$
```

### 1.3.4 dlist_push_back

```c
int dlist_push_back(struct dlist *list, int element);
```

Add `element` to the back of `list`. In case of failure return `0`. Otherwise, return `1`.

#### 1.3.5 dlist_size

```
size_t dlist_size(const struct dlist *list);
```

Return the size of `list`.

### 1.4 Threshold 2

#### 1.4.1 dlist_get

```
int dlist_get(struct dlist *list, size_t index);
```

Return the element at `index`. If the `index` is not within list, return `-1`.

#### 1.4.2 dlist_insert_at

```
int dlist_insert_at(struct dlist *list, int element, size_t index);
```

Insert `element` at `index`. If `index` is not within `list`, or if any argument is invalid, return `-1`. Otherwise, return `1`. Insertion at the index `size` is considered valid.

#### 1.4.3 dlist_find

```
int dlist_find(const struct dlist *list, int element);
```

Search for the first occurrence of `element`. If found, the index is returned, return **-1** otherwise.

#### 1.4.4 dlist_remove_at

```
int dlist_remove_at(struct dlist *list, size_t index);
```

Remove the integer at `index` from `list` and return it.

If the index is not within `list`, return **-1**.

### 1.5 Threshold 3

#### 1.5.1 dlist_map_square

```
void dlist_map_square(struct dlist *list);
```

Replace each element in `list` by its square.

### 1.5.2 dlist_reverse

```
void dlist_reverse(struct dlist *list);
```

Reverse the order of the elements in `list`.

### 1.5.3 dlist_split_at

```
struct dlist *dlist_split_at(struct dlist *list, size_t index);
```

Split `list` in two at `index`. `list` must keep the first half and the second half must be returned. The element at `index` belongs to the second half.

If `index` is bigger than the size of the list, or if anything goes wrong, you must return `NULL`. An empty list cannot be split, you just need to return a newly allocated dlist.

### 1.5.4 dlist_concat

```
void dlist_concat(struct dlist *list1, struct dlist *list2);
```

Append `list2` to `list1`. `list2` must be emptied.

*It is my job to make sure you do yours.*