Exercises — Dlist

version **#7be580532266ed398481e31366afcc24b1950c2a**

The way is lit. The path is clear.
We require only the strength to follow it.

Assistants C/Unix 2023 <assistants@tickets.assistants.epita.fr>

Copyright

The use of this document must abide by the following rules:

- ▷ You downloaded it from the assistants' intranet.*
- ▷ This document is strictly personal and must **not** be passed onto someone else.
- ▷ Non-compliance with these rules can lead to severe sanctions.

Contents

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

File Tree

```
dlist/
├─ *   (to submit)
├─ Makefile   (to submit)
└─ dlist.h
```

Makefile

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

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

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

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

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

Compilation : Your code must compile with the following flags

- -std=c99 -pedantic -Werror -Wall -Wextra -Wvla

Main function : None

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_item
{
    int data;
```

(continues on next page)

(continued from previous page)

```
    struct dlist_item *next;
    struct dlist_item *prev;
};

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

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

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.
- An empty list is a list where `size` is 0, and `head` and `tail` are `NULL`.
- 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.

  Be careful!

  This exercise is followed by a second part named `dlist_advanced`. Some instruction given in this preambule are important only for this next part.

3 Threshold 1

3.1 dlist_init

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

Return a new allocated empty list.

3.2 dlist_push_front

```
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`.

3.3 dlist_print

```
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:

```
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$
```

3.4 dlist_push_back

```
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.

3.5 dlist_size

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

Return the size of list.

4 Threshold 2

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.

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.

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.

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**.

Tips

You are responsible of the allocation and the deallocation of the `dlist_item`.

5 Threshold 3

5.1 dlist_map_square

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

Replace each element in `list` by its square.

5.2 dlist_reverse

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

Reverse the order of the elements in `list`.

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.

5.4 dlist_concat

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

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

The way is lit. The path is clear. We require only the strength to follow it.