Exercises — Dlist Advanced

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_advanced/
├─ *   (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

Be careful!

This exercise is the next step after having completed the *dlist* exercise. You need to follow the same specification described in the *Preambule* part of *dlist*'s subject.

In this step, you will manipulate doubly linked lists in a more advanced way.

1.1 Threshold 4

1.1.1 dlist_clear

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

Empty `list` by freeing its nodes. However, `list` must not be freed.

1.1.2 dlist_shift

```
void dlist_shift(struct dlist *list, int offset);
```

Shift the list (by looping) of |`offset`| elements (where $|x|$ is the absolute value of x). This shift is a left shift if `offset` is negative and is a right shift if `offset` is positive.

Example:

```
list1 = [1,2,3,4,5]
/* Shift 1 */
list1 = [5,1,2,3,4]

list2 = [2,4,6,8,10]
/* Shift -7 */
list2 = [6,8,10,2,4]
```

1.1.3 dlist_add_sort

```
int dlist_add_sort(struct dlist *list, int element);
```

Add `element`, keeping `list` sorted in increasing order. This will only be tested with an already sorted list. If any argument is invalid, return `-1`. Otherwise, return `1`.

1.1.4 dlist_remove_eq

```
int dlist_remove_eq(struct dlist *list, int element);
```

Remove the first element equal to `element`. Return `1` if an element has been removed, `0` otherwise.

1.1.5 dlist_copy

```
struct dlist *dlist_copy(const struct dlist *list);
```

Return a **deep** copy of `list`: each element of the list must be copied. If anything goes wrong, return `NULL`.

1.2 Threshold 5

1.2.1 dlist_sort

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

Sort `list` in increasing order with the algorithm of your choice. A simple bubble sort should do the trick.

1.2.2 dlist_merge

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

Merge two sorted lists. The result of the merge must be a sorted list. `list2` must be emptied. The lists are sorted in increasing order.

Before:

```
list1 = [1,4,7,8]
list2 = [2,5,6,7]
```

After:

```
List1 = [1,2,4,5,6,7,7,8]
List2 = []
```

1.3 Threshold 6

1.3.1 dlist_levenshtein

```
unsigned int dlist_levenshtein(struct dlist *list1, struct dlist *list2);
```

Returns the levenshtein distance between `list1` and `list2`.

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