Семинар #6: Динамический массив. Домашнее задание.

Задача 1. Новые функции для динамического массива

В папке `dynarray` лежит реализация динамического массива. Вам нужно написать ещё несколько функций для работы с этим динамическим массивом и протестировать их. Функции нужно написать в файлах `dynarray.h` и `dynarray.c`, а код для тестирования в файле `main.c`.

1.1. `int pop_back(Dynarray* pd)`

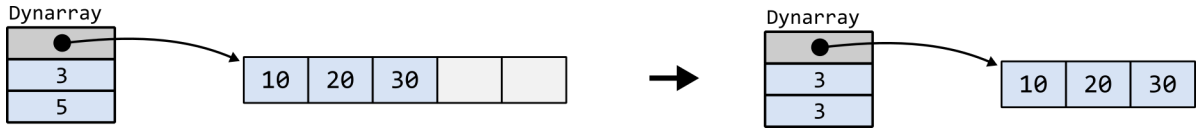
Функция должна будет удалять последний элемент массива и возвращать его. Вместимость массива в этой функции меняться не должна. В случае если массив пуст и удалять нечего функция должна напечатать сообщение об ошибке и выйти из программы.

1.2. `void resize(Dynarray* pd, size_t new_size)`

Функция должна будет изменять размер массива на `new_size`. Если `new_size` будет меньше, чем старый размер, то размер массива должен уменьшиться. При увеличении размера новые элементы массива должны быть равны нулю.

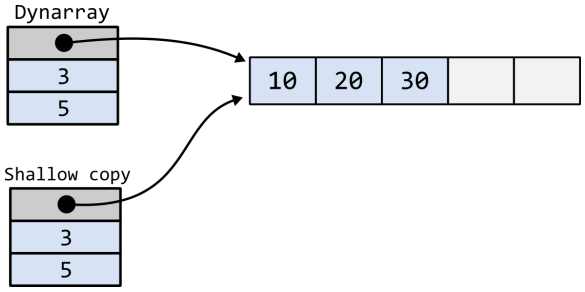
1.3. `void shrink_to_fit(Dynarray* pd)`

Функция должна делать вместимость массива равной её размеру.



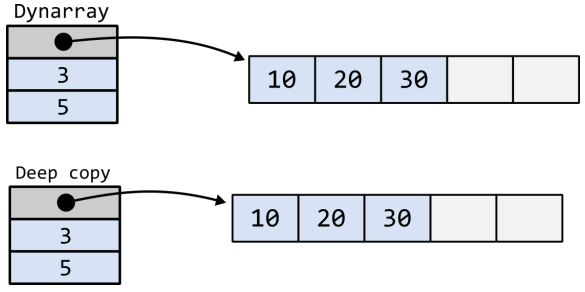
1.4. Поверхностная копия: `Dynarray shallow_copy(Dynarray* pd)`

Создаёт поверхностную копию динамического массива. При поверхностном копировании копируется только сам объект, а все ресурсы, связанные с этим объектом не копируются.



1.5. Глубокая копия: `Dynarray deep_copy(const Dynarray* pd)`

Создаёт глубокую копию динамического массива. При глубоком копировании копируется и сам объект и все ресурсы, связанные с этим объектом.



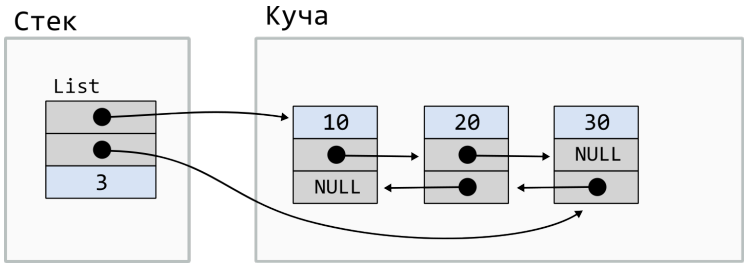
Задача 2. Двусвязный список

Напишите свою реализацию структуры данных двусвязный список, хранящий элементы типа `int`. Узел такого списка будет выглядеть так:

```
struct node
{
    int value;
    struct node* next;
    struct node* prev;
};
typedef struct node Node;
```

Структура для самого двусвязного списка и расположение двусвязного списка в памяти:

```
struct list
{
    Node* head;
    Node* tail;
    size_t size;
};
typedef struct list List;
```



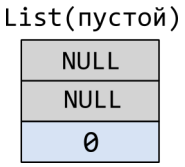
Поля структуры `List`:

- `head` – указатель на первый узел двусвязного списка
- `tail` – указатель на последний узел двусвязного списка
- `size` – количество элементов в двусвязном списке

Вам нужно будет написать и протестировать следующие функции для работы с двусвязным списком. Реализация списка должна находиться в файлах `list.h` и `list.c`. Заголовочный файл должен подключаться к файлу `main.c`, в котором и должно происходить тестирование списка.

2.1. `List init(size_t n)`

Возвращает двусвязный список из `n` нулевых элементов. Если `n` равно нулю, то возвращает пустой двусвязный список, который должен выглядеть вот так:



2.2. `void print(const List* pl)`

Печатает все элементы списка.

2.3. `void push_back(List* pl, int value)`

Добавляет элемент в конец списка.

2.4. `int pop_back(List* pl)`

Удаляет элемент из конца списка и возвращает его.

2.5. `void push_front(List* pl, int value)`

Добавляет элемент в начало списка.

2.6. `int pop_front(List* pl)`

Удаляет элемент из начала списка и возвращает его.

2.7. `Node* erase(List* pl, Node* p)`

Указатель `p` указывает на один из узлов списка. Функция должна удалять узел, на который указывает `p`, и возвращать указатель на узел, следующий после удалённого. В случае удаления последнего узла функция должна возвращать `NULL`.

2.8. `void splice(List* plist, Node* p, List* pother)`

Переносит все элементы из списка на который указывает `pother` в список на который указывает `plist`. Новые элементы должны быть вставлены сразу перед узлом на который указывает `p`. Порядок элементов не должен измениться. После выполнения этой операции список на который указывает `pother` должен стать пустым.

2.9. `void destroy(List* pl)`

Удаляет все элементы списка и зануляет все поля структуры `List`.

2.10. `void advance(Node** pp, size_t n)`

Функция должна принимать адрес указателя на один из узлов связного списка и изменяет этот указатель так, чтобы он указывал на `n` узлов вперёд. Если количество узлов до конца списка меньше, чем `n` то функция должна изменить значение указателя на `NULL`.

2.11. Тестирование

Протестируйте вашу реализацию списка, используя следующий код:

```
#include <stdio.h>
#include "list.h"
int main()
{
    List a = init(0);

    for (int i = 0; i < 5; ++i)
        push_back(&a, 10 * (i + 1));
    for (int i = 0; i < 5; ++i)
        push_front(&a, 100 * (i + 1));
    print(&a);                          // Напечатает 500 400 300 200 100 10 20 30 40 50
                                        //
    printf("%i\n", pop_front(&a));      // Напечатает 500
    printf("%i\n", pop_back(&a));       // Напечатает 50
    print(&a);                          // Напечатает 400 300 200 100 10 20 30 40
                                        //
    Node* p = a.head;                   //
    advance(&p, 3);                     //
    p = erase(a, p);                    //
    print(&a);                          // Напечатает 400 300 200 10 20 30 40
                                        //
    List b = init(0);                   //
    for (int i = 0; i < 3; ++i)         //
        push_back(&a, 1000 * (i + 1));  //
    splice(&a, p, &b);                  //
                                        //
    print(&a);                          // Напечатает 400 300 200 1000 2000 3000 10 20 30 40
    print(&b);                          // Ничего не напечатает
}
```

Задача 3. Счет

Напишите программу, которая будет вести себя по-разному в зависимости от того какая опция была передана при компиляции. Программа должна печатать числа от `1` до значения передаваемого на этапе компиляции макроса `COUNT`. То есть, если скомпилировать программу так:

```
gcc -DCOUNT=5 main.c
```

то программа при запуске должна напечатать числа от `1` до `5`:

```
1 2 3 4 5
```

В случае же если опции `-DCOUNT` при компиляции программы передано не было, например при такой компиляции:

```
gcc main.c
```

программа при запуске должна напечатать:

```
No Count!
```

Необязательные задачи (не входят в ДЗ, никак не учитываются)

Задача 1. Очередь

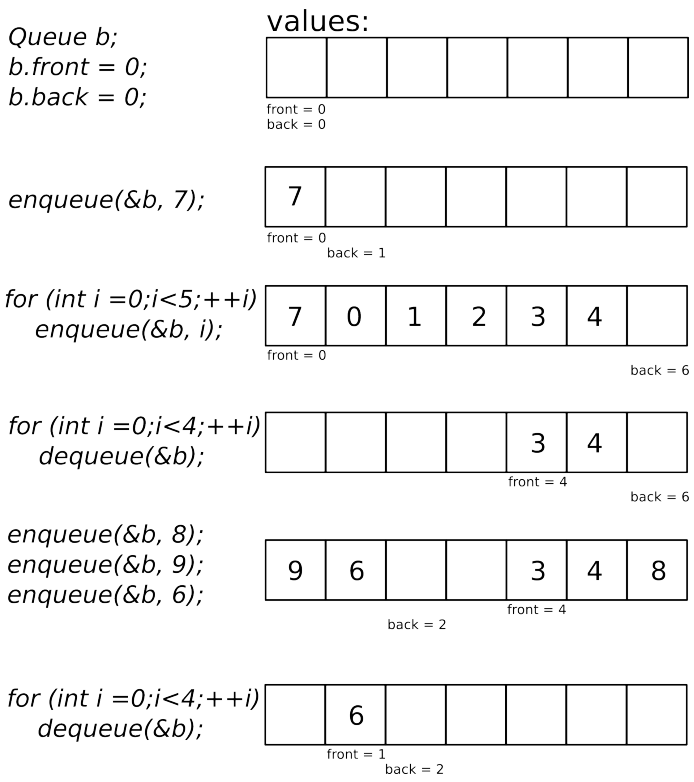
```
#define CAPACITY 7
typedef int Data;

struct queue
{
    int front;
    int back;
    Data values[CAPACITY];
};
typedef struct queue Queue;

// ......

int main()
{
    Queue a;
    queue_init(&a);
    enqueue(&a, 100);
    for (int i = 0; i < 20; ++i)
    {
        enqueue(&a, i);
        dequeue(&a);
    }
    enqueue(&a, 200);
    queue_print(&a);
}
```

Очередь — абстрактный тип данных с дисциплиной доступа к элементам «первый пришёл — первый вышел».
Реализация с помощью массива:



Задача #3: Очередь на основе статического массива:

1. Написать функцию `void queue_init(Queue* q)`, которая будет задавать начальные значения полей `front` и `back`.
2. Написать функцию `void enqueue(Queue* q, Data x)` - добавляет `x` в очередь. Для эффективной реализации очереди, нужно использовать как можно меньше операций и как можно эффективней использовать выделенную память. Поэтому, при заполнении массива, если начало массива свободно, то элементы можно хранить там. (смотрите рисунок)
3. Написать функцию `Data dequeue(Queue* q)` - удаляет элемент из очереди и возвращает его. Для эффективной реализации очереди сдвигать оставшиеся элементы не нужно. Вместо этого можно просто увеличить поле `front`.
4. Написать функцию `int queue_is_empty(const Queue* q)`, которая возвращает `1` если очередь пуста и `0` иначе.
5. Написать функцию `int queue_get_size(const Queue* q)`, которая возвращает количество элементов.
6. Написать функцию `int queue_is_full(const Queue* q)`, которая возвращает `1` если очередь заполнена и `0` иначе. Очередь считается полной, если `size == capacity - 1`.
7. Написать функции `Data queue_get_front(const Queue* q)` и `Data queue_get_back(const Queue* q)`, которые возвращают элементы, находящиеся в начале и в конце очереди соответственно, но не изменяют очередь.

8. Написать функцию `void queue_print(const Queue* q)`, которая распечатывает все элементы очереди.

9. Что произойдёт, если вызвать `enqueue` при полной очереди или `dequeue` при пустой? Обработайте эти ситуации. Программа должна печатать сообщение об ошибке и завершаться с аварийным кодом завершения. Чтобы завершить программу таким образом можно использовать функцию `exit` из библиотеки `stdlib.h`.

10. Протестируйте очередь на следующих тестах:

    (a) В очередь добавляется 4 элемента, затем удаляется 2. Вывести содержимое очереди с помощью `queue_print()`

    (b) В очередь добавляется очень много элементов (больше чем `CAPACITY`). Программа должна напечатать сообщение об ошибке.

    (c) В очередь добавляется 3 элемента, затем удаляется 2, затем добавляется очень много элементов (больше чем `CAPACITY`). Программа должна напечатать сообщение об ошибке.

    (d) В очередь добавляется 3 элемента, затем удаляется 4. Программа должна напечатать сообщение об ошибке.

    (e) В очередь добавляется 2 элемента, затем выполняется следующий цикл:

    ```
    for (int i = 0; i < 10000; ++i)
    {
        enqueue(&a, i);
        dequeue(&a);
    }
    ```

    Вывести содержимое очереди с помощью `queue_print()`

Задача #4: Очередь на основе динамического массива:
Описание такой очереди выглядит следующим образом:

```
struct queue
{
    int capacity;
    int front;
    int back;
    Data* values;
};
typedef struct queue Queue;
```

1. Скопируйте код очереди со статическим массивом в новый файл и измените описание структуры как показано выше. Макрос `CAPACITY` больше не нужен, его можно удалить.

2. Измените функцию `void queue_init(Queue* q)` на `void queue_init(Queue* q, int initial_capacity)`. Теперь она должна присваивать `capacity` начальное значение `initial_capacity` и выделять необходимую память под массив `values`.

3. Измените функцию `void enqueue(Queue* q)`. Теперь, при заполнении очереди должно происходить перевыделение памяти с помощью функции `realloc`. Заполнение очереди достигается когда размер очереди становится равным `capacity - 1` (а не `capacity`, потому что при полном заполнении вместимости `front` будет равняться `back` и мы не сможем понять полная эта очередь или пустая). После перевыделения нужно переместить элементы массива на новые места и изменить `front` и `back`. Если `front != 0`, то нужно переместить элементы массива от `front` до конца старого массива `values` в конец нового массива `values`. (смотрите рисунок ниже)

4. Добавьте функцию `void queue_destroy(Queue* q)`, которая будет освобождать память, выделенную под массив `values`.

5. Протестируйте очередь: в очередь добавляется много элементов ($\gg 10^3 >$ `initial_capacity`). Программа **не** должна напечатать сообщение об ошибке (если только совокупный размер элементов не превышает размер доступной оперативной памяти).

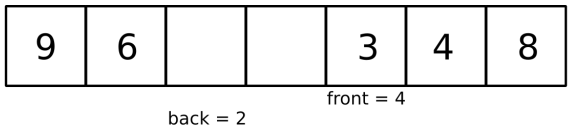
6. В случае, если `malloc` или `realloc` не смогли выделить запрашиваемый объём памяти (например, по причине того, что этот объём больше, чем вся доступная оперативная память или по какой-нибудь иной причине), то они возвращают значение `NULL`. Программа должна это учитывать и завершаться с ошибкой, если нельзя выделить нужный объём памяти.

Схема перевыделения памяти для очереди на основе динамического массива:

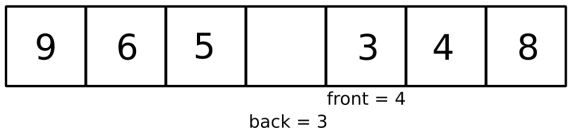
Очередь будет считаться заполненной:

- Если `front == 0`, а `back == capacity - 1`
- Или если `front != 0`, а `front - back == 1`. (А не `front - back == 0`, потому что при полном заполнении вместимости `front` будет равняться `back` и мы не сможем понять полная эта очередь или пустая).

Когда очередь заполнена и мы хотим добавить в неё ещё один элемент, то её нужно увеличить. Делается это так, как представлено на схеме ниже:



enqueue(&b, 5);



enqueue(&b, 7);

- Если `front == 0`, то нужно просто увеличить очередь с помощью `realloc`.
- Если `front != 0`, то нужно ещё и перекопировать хвост очереди в конец и изменить `front`.