Семинар #8: Стек и Очередь. Домашнее задание.

Абстрактные типы данных: Стек, очередь, дек, очередь с приоритетом

Абстракстный тип данных (АТД) - это математическая модель для типов данных, которая задаёт поведение этих типов, но не их внутреннею реализацию.

Стек (Stack) - это АТД, который представляет собой коллекцию элементов, менять которые можно только с помощью двух операций:

- `push` - добавить элемент в стек.
- `pop` - извлечь из стека последний добавленный элемент.

Таким образом, поведение стека задаётся этими двумя операциями. Так как стек - это абстрактный тип данных, то его внутренняя реализация на языке программирования может быть самой разной. Стек можно сделать на основе статического массива, на основе динамического массива(`malloc`/`free`) или на основе связного списка. Внутренняя реализация не важна, важно только наличие операций `push` и `pop`.
Не нужно путать абстракстный тип данных стек с сегментом памяти стек.

Очередь (Queue) - это АТД, который представляет собой коллекцию элементов, менять которые можно только с помощью двух операций:

- `enqueue` - добавить элемент в очередь.
- `dequeue` - извлечь из очереди первый добавленный элемент из оставшихся.

`../images/stack_queue.png`

Дек (Deque = Double-ended queue) - это АТД, который представляет собой коллекцию элементов, менять которые можно только с помощью четырёх операций:

- `push_back` - добавить элемент в конец.
- `push_front` - добавить элемент в начало.
- `pop_back` - извлечь элемент с конца.
- `pop_front` - извлечь элемент с начала.

Очередь с приоритетом (Priority Queue) - это АТД, который представляет собой коллекцию элементов, менять которые можно только с помощью двух операций:

- `insert` - добавить элемент.
- `extract_best` - извлечь из очереди элемент с наибольшим приоритетом.

То, что будет являться приоритетом может различаться. Это может быть как сам элемент, часть элемента (например, одно из полей структуры) или другие данные, подаваемые на вход операции `insert` вместе с элементом. В простейшем случае, приоритетом является сам элемент (тогда очередь с приоритетом просто возвращает максимальный элемент) или сам элемент со знаком минус (тогда очередь с приоритетом возвращает минимальный элемент).

Реализация стека на основе статического массива

В файле `stack/1stack.c` представлена реализация стека на основе статического массива.
Основные моменты такой реализации:

- Константа `CAPACITY` задаёт вместимость стека. Эта величина будет постоянна во время выполнения программы. Если, вдруг, во время выполнения программы нам понадобится больше места в стеке, то мы ничего сделать не сможем и придётся писать сообщение об ошибке и завершать программу. В то же время, если в этом стеке будет мало элементов, то он всё-равно будет потреблять память соответствующую `CAPACITY` элементам. Эти недостатки решаются в реализации на основе динамического массива.

- Строка

  ```
  typedef int Data;
  ```

  создаёт новое имя для типа `int`. Благодаря этому вместо `int` можно писать `Data`. Это нужно для того, чтобы можно было легко менять тип элемента, который хранит стек. Если, например, мы захотим хранить в стеке не `int`-ы, а `char`-ы или что-то другое, то мы просто изменим эту строку. Однако, если нам понадобится в одной программе использовать 2 стека с разными типами, то такое не пройдёт и нам придётся дублировать код для разных стеков. От этого недостатка можно избавиться в языке `C++` с помощью шаблонов.

- Структура `Stack` содержит целое `size` – количество элементов в стеке и массив `values` - массив размера `CAPACITY`, в первых `size` ячейках которого будут хранится элементы стека.

- Функция `stack_init` задаёт поля структуры стека для начала работы с ним. В данном случае она просто зануляет `n`. В других реализациях аналогичные функции будут иметь более сложный вид.

- Функция `stack_push` добавляет элемент `x` в стек. Если стек уже заполнен, то программа завершает своё выполнение с ошибкой. Стек передаётся по указателю, так как он будет меняться внутри функции.

- Функция `stack_pop` удаляет последний элемент стека и возвращает его. Если стек пуст, то программа завершает своё выполнение с ошибкой.

- Функция `stack_get` возвращает последний элемент стека. Стек передаётся по `const`-указателю, так как стек не будет меняться внутри этой функции. А передавать стек в функцию по значению было бы слишком затратно, так как происходило бы копирование всего стека.

Реализация стека на основе динамического массива

В файле `stack/2stack_dynamic.c` представлена реализация стека на основе динамического массива.
Основные моменты такой реализации:

- Структура `Stack` содержит `capacity` – вместимость стека, `size` – количество элементов в стеке и указатель `values`. Указатель `values` будет указывать на участок памяти, выделенной с помощью `malloc`. Размеры выделенной памяти будут храниться в поле `capacity`.
- Функция `stack_init` теперь должна задать `size` и `capacity`, а также выделить начальную область памяти под стек. Код выделения памяти выглядит следующим образом:

```
s->values = (Data*)malloc(s->capacity * sizeof(Data));
if (s->values == NULL)
{
        printf("Error! Could not allocate memory.\n");
        exit(1);
}
```

  В первой строке мы пытаемся выделить необходимое количество памяти. При этом иногда (очень редко) у `malloc`-а может не получиться выделить нужное количество памяти (например если вся память кончилась). В этом случае `malloc` возвращает `NULL`. Если это происходит, то мы должны завешить программу.

- Функция `stack_push` добавляет элемент в стек. Но перед тем, как это сделать функция проверит, если в стеке место. Если места нет, то функция должна увеличить вместимость динамического массива. Происходит это следующим образом:

```
s->capacity *= 2;
Data* temp = (Data*)realloc(s->values, s->capacity * sizeof(int));
if (temp == NULL)
{
        printf("Error! Could not reallocate memory\n");
        free(s->values);
        exit(1);
}
else
{
        s->values = temp;
}
```

  В второй строке мы пытаемся выделить больше памяти. Если перевыделение прошло успешно, то указатель `temp` будет хранить адрес новой памяти. При этом старая память автоматически освободится. Если же перевыделение не сработало(память кончилась), то `temp` будет равным `NULL`.

Задачи

Задача #1: Размер и вместимость: Напишите программу, которая будет создавать стек(динамический) вместимости `7` и добавлять в него последовательно `200` элементов. При каждом добавлении элемента печатайте размер и вместимость стека.
Задача #2: Скобочки: Написать программу которая будет считывать последовательность скобочек и печатать `Yes` или `No` в зависимости от того является ли эта последовательность допустимой. Используйте одну из реализаций стека. Для считывания строки: `scanf("%s", str);`

вход	выход
`()`	Yes
`{[()]}`	Yes
`)))))`	No
`([)]`	No
`[{}()]`	Yes

вход	выход
`)(`	No
`(((((((())))))))`	Yes
`{{{{`	No
`{[([][)[{}])][()]}`	Yes
`]`	No

Очередь

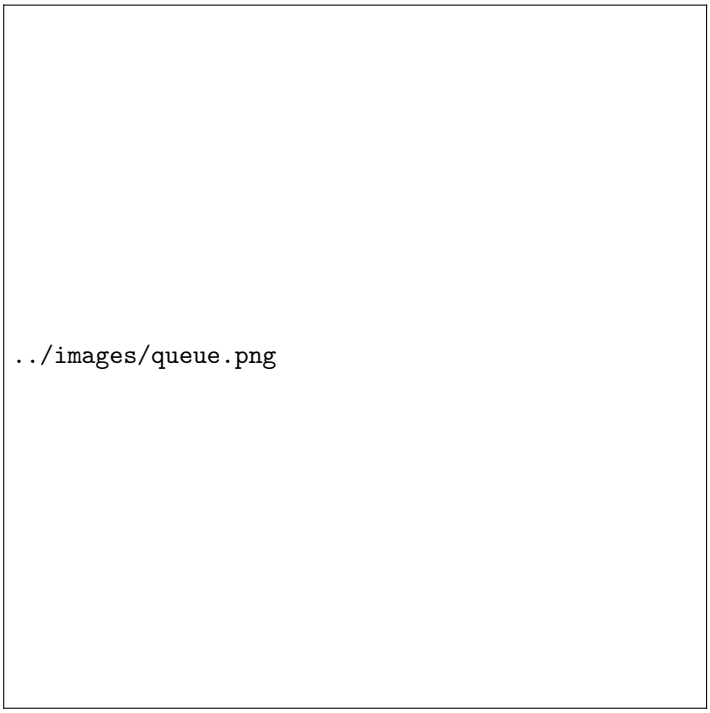
```
#define CAPACITY 7
typedef int Data;

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).

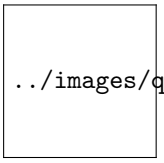
6. В случае, если `malloc` или `realloc` не смогли выделить запрашиваемый объём памяти (например, по причине того, что этот объём больше, чем вся доступная оперативная память или по какой-нибудь иной причине), то они возвращают значение `NULL`. Программа должна это учитывать и завершаться с ошибкой,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

Схема певыделения 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:



- 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`.