acm.mipt.ru

олимпиады по программированию на Физтехе

Раздел «Язык Си» . CoffeDList :

Задачи: **list_2** и **list_3** из контекста.

Предполагается, что тема между стеком и двусвязанным списком - это Интерфейс и реализация, односвязный список и стек на основе односвязного списка

Список - это динамическая структура данных, состоящая из **узлов.** Каждый узел содержит данные и ссылки на один соседний узел (односвязный список) или два соседних узла (двусвязный список).

Список может быть с открытым концом (первый и последний элемент обычно указывают на NULL), а может быть кольцевым или *циклическим* (хоровод).

Если в циклическом списке явно выделен один элемент, как в бусах есть бусины и есть отдельно замок, то этот выделенный элемент называется барьерным.

Циклический список с барьерным элементом имеет самую простую реализацию.

Договоримся, что

- **начало списка** - это замок->next.
- **конец списка** - это замок->prev.

Двусвязанный список

Пусть в нашем списке хранятся данные - целые числа.

```
typedef int Data;
```

Для создания узла двусвязанного списка определим структуру

```
struct Node {
    Data data;             // данные
    struct Node * prev;    // указатель на предыдущий узел списка
    struct Node * next;    // указатель на следующий узел списка
};
```

Создаем заглушку и пишем функцию печати

Для понимания, как устроен список и как его перебрать с начала до конца, сделаем список из узлов "руками" и попробуем пройтись по ним.

```
#include <stdio.h>

typedef int Data;       // покажем список на примере целых чисел

struct Node {           // один узел списка
    Data data;          // данные
    struct Node * prev; // указатель на предыдущий узел
    struct Node * next; // указатель на следующий узел
};

int main() {
    struct Node
        z,              // замковый элемент, данные - мусор (не знаем какие)
        a, b, c, d, f;     // прочие узлы
```

```
    // сделаем список из чисел 5, 7, -3
    // замковый элемент, данные - мусор
    z.next = &a;
    z.prev = &c;

    a.data = 5;
    a.next = &b;
    a.prev = &z;

    b.data = 7;
    b.next = &c;
    b.prev = &a;

    c.data = -3;
    c.next = &z;
    c.prev = &b;

    // для печати данных из списка нужно выполнить код
    // данные в z не печатаем, там мусор
    printf("%d ", a.data);  // 5
    printf("%d ", b.data);  // 7
    printf("%d ", c.data);  // -3
    printf("\n");

    // или реализуем функцию печати списка и вызовем ее
    // list_print(&z);

    return 0;
}
```

Напишем функцию печати. Мы не знаем сколько элементов будет в списке, поэтому нужно написать цикл с условием окончания.

Начинать печать нужно с первого элемента *после* замка (у нас – узел a).

Указатель p пусть указывает по очереди на узлы от a до c (помним, что в функции печати у нас нет таких переменных, есть только ссылка на замковый элемент, которая хранится в переменной list).

Продолжать печать – пока p указывает на "бусины", а прекращаем, когда p указывает на "замок".

Рисунок: p проходит указателем от a до c.

```
void list_print(struct Node * list) {
    struct Node * p;
    for (
        p = list->next;     // первая бусина - после замка
        p != list;          // печатаем только бусины, стоп если дошли до замка
        p = p->next
    ) {
        printf("%d ", p->data); // печать чисел
    }
    printf("\n");
}
```

При отладке нам захочется иметь полную информацию о списке, со всеми указателями. Модифицируем функцию печати так, чтобы при определенном макросе LIST_DBG была полная отладочная печать *всех* элементов.

```
void list_print(struct Node * list) {
    struct Node * p;
#ifdef LIST_DBG
    printf("LIST:\tthis=%p prev=%p next=%p\n", list, list->prev, list->next); // замок
#endif
    for (
        p = list->next;     // первая бусина - после замка
        p != list;          // печатаем только бусины, стоп если дошли до замка
        p = p->next
    ) {
#ifdef LIST_DBG
        printf("%d\tthis=%p prev=%p next=%p\n", p->data, p, p->prev, p->next); // печать чисел
#else
        printf("%d ", p->data); // печать чисел
#endif
```

```
    }
    printf("\n");
}
```

Напомним, чтобы запустить программу с определенным макросом LIST_DBG нужно в коде программы написать

```
#define LIST_DBG
```

или при компиляции указать дополнительную опцию компилятора -D со значением LIST_DBG:

```
gcc -Wall -Wextra -DLIST_DBG dlist0.c
```

List API

API - application programming interface - набор функций для работы.

Для работы со списком напишем функции:

```
// делает список пригодным для работы
void list_init(struct Node * list);

// вставка и удаление элемента БЕЗ выделения памяти (операция с узлами)
void list_insert(struct Node * list, struct Node * t);
void list_insert_before(struct Node * list, struct Node * t);
void list_remove(struct Node * t);

// вставка элемента С выделением памяти
struct Node * list_push_front(struct Node * list, Data d);
struct Node * list_push_back(struct Node * list, Data d);

// удаление 1 элемента С освобождением памяти
Data list_pop_front(struct Node * list);
Data list_pop_back(struct Node * list);
Data list_delete(struct Node * t);

// удаление всех элементов, кроме "замка"
void list_clear(struct Node * list);

// вспомогательные функции печати и проверки на пустоту
void list_print (struct Node * list);
int list_is_empty(struct Node * list);
```

Вставка одного элемента БЕЗ выделения памяти

Попробуем реализовать функцию

```
void list_insert(struct Node * list, struct Node * t);
```

Для этого попытаемся в нашей "заглушке" dlist0.c в функции main написать код, который вставляет узел d после узла a.

Добавим этот код в функцию main:

```
    d.data = 10;
    // вставим узел d после узла a
    // вставим узел d после узла a
    a.next = &d;        // запишем адрес 700 в поле next узла с адресом 100
    b.prev = &d;        // запишем адрес 700 в поле prev узла с адресом 200
    d.next = &b;        // запишем адрес 200 в поле next узла с адресом 700
    d.prev = &a;        // запишем адрес 100 в поле prev узла с адресом 700
    // list_insert(&a, &d);

    // убедимся, что узлы вставились верно
    list_print(&z);         // 5 10 7 -3
```

Обязательно запустите печать с отладочной информацией и убедитесь, что все ссылки поставлены правильно.

Вопрос: Без проверки ссылок с помощью отладочной печати, гарантирует ли печать чисел 5 10 7 -3, что вставка узла реализована правильно?

list_insert

Глядя на "заглушечный" код, реализуем функцию list_insert. Меняем:

- a. на p->
- b. на n->
- &d на t

```
// вставляем узел t в список после узла p
void list_insert(struct Node * p, struct Node * t) {
    // объявим новую переменную n - указатель на следующий узел списка после p
    // (в заглушке это был узел b)
    struct Node * n = p->next;

    // вставим узел t после узла p
    p->next = t;        // запишем адрес 700 в поле next узла с адресом 100
    n->prev = t;        // запишем адрес 700 в поле prev узла с адресом 200
    t->next = n;        // запишем адрес 200 в поле next узла с адресом 700
    t->prev = p;        // запишем адрес 100 в поле prev узла с адресом 700
}
```

Заменяем "заглушечный" код в main на вызов функции list_insert(&a, &d);

Вставка одного элемента d в начало списка – это вставить элемент сразу после "замка" z:

```
list_insert(&z, &d);
```

Как мы видим, одна функция позволяет и вставлять узел сразу после произвольного узла, и вставлять в начало списка.

Можно написать эту функцию без введения дополнительной переменной n:

```
// вставляем узел t в список после узла p
void list_insert(struct Node * p, struct Node * t) {
    // вставим узел t после узла p
    p->next->prev = t;        // запишем адрес 700 в поле prev узла с адресом 200
    t->next = p->next;        // запишем адрес 200 в поле next узла с адресом 700
    p->next = t;        // запишем адрес 700 в поле next узла с адресом 100
    t->prev = p;        // запишем адрес 100 в поле prev узла с адресом 700
}
```

Сравните две реализации функции. В какой проще сделать ошибку при реализации? Какая функция не зависит от порядка строк кода (заполнения полей next и prev в узлах)?

list_insert_before

Теперь напишем вставку узла f ПЕРЕД узлом b. Можно аналогично писать

```
// вставляем узел t в список ПЕРЕД узлом n
void list_insert_before(struct Node * n, struct Node * t) {
    // объявим новую переменную p - указатель на ПРЕДЫДУЩИЙ узел списка перед n
    struct Node * p = n->prev;

    // далее код не изменился:
    // вставим узел t после узла p
    p->next = t;        // запишем адрес 700 в поле next узла с адресом 100
    n->prev = t;        // запишем адрес 700 в поле prev узла с адресом 200
    t->next = n;        // запишем адрес 200 в поле next узла с адресом 700
    t->prev = p;        // запишем адрес 100 в поле prev узла с адресом 700
}
```

Но это неверный стиль программирования.

Потому что мы можем сделать тут ошибку (или написать list_insert_before верно, а ошибка будет в list_insert). Или обе функции будут содержать *разные* ошибки.

Проще, если обе функции будут работать или обе правильно, или обе ОДИНАКОВО неправильно (быстрее заметим ошибку, меньше исправлять).

Для этого будем **использовать код повторно.**

"Вставить узел t ПЕРЕД узлом n" означает "Вставить узел t ПОСЛЕ узла n->prev"

```
// вставляем узел t в список перед узлом n
void list_insert_before(struct Node * n, struct Node * t) {
    list_insert(n->prev, t);
}
```

Как видим, вероятность ошибки минимальная.

list_remove

Операция удаления одного элемента из списка - это обратная операция вставке элемента в список. Напишем ее сразу в виде функции (проще ее реализовать через локальные переменные p и n - указатели на предыдущий и следующий после t узлы в списке:

```
// удаляем узел t из списка
void list_remove(struct Node * t) {
    struct Node * p = t->prev;
    struct Node * n = t->next;
    p->next = n;
    n->prev = p;
}
```

Проверьте, что функция работает правильно.

Инициализация списка

Мы делали "заглушку" в main - создавали список из узлов a, b, c вручную. Сделаем пустой список так, чтобы в него можно было добавлять узлы с помощью list_insert и list_insert_before.

Пустой список тоже должен быть циклическим и содержать барьерный элемент ("замок"). Он не должен содержать других элементов.

То есть в main код должен быть таким:

```
struct Node z;
struct Node * list = &z;
list_init(list);
```

В пустом циклическом списке у "замка" следующий и предыдущий элементы - сам же "замок". На схеме получается фигура "руки в боки".

Рисунок: пустой список и руки в боки.

```
// инициализация списка: следующий и предыдущий для "замка" - сам "замок"
void list_init(struct Node * list) {
    list->next = list;
    list->prev = list;
}
```

Глядя на list_init реализуйте функцию list_is_empty

```
int list_is_empty(struct Node * list) {
    // функция возвращает ИСТИНУ, если список пустой (только "замок"), иначе возвращает ЛОЖЬ
    // тут нужно написать 1 строку кода
}
```

Перепишем функцию main для тестирования функций.

```
int main() {
    struct Node z;              // "замок"
    struct Node * list = &z;    // указатель на список
    struct Node a = {5}, b = {7}, c = {-3}, d = {10}, f = {22};

    list_init(list);
    list_print(list);       // ничего не печатается

    printf("is_empty = %d\n", list_is_empty(list));   // ДА

    list_insert(list, &c);  // очень похоже на стек?
    printf("is_empty = %d\n", list_is_empty(list));   // НЕТ
    list_insert(list, &b);
    list_insert(list, &a);
    list_print(list);       // 5 7 -3

    printf("is_empty = %d\n", list_is_empty(list));   // НЕТ

    list_insert(list->next, &d);
    list_insert_before(list, &f);
    list_print(list);       // 5 10 7 -3 22
```

```
    list_remove(&d);
    list_remove(&f);
    list_print(list);       // 5 7 -3

    return 0;
}
```

Заметьте, что нигде выше память динамически не выделялась и не освобождалась.

Вставка и удаление данных + работа с памятью

Напишем следующий блок функций, который вставляет данные и удаляет данные из списка, **выделяя и освобождая память динамически.**

Сравним прототипы функций list_insert и list_push. Видно, что в первую функцию передается готовый узел t, а во вторую - только данные data.

```
void list_insert(struct Node * list, struct Node * t);
struct Node * list_push_front(struct Node * list, Data data);
```

Что возвращает list_push_front? Указатель на вновь созданный узел.

Для начала напишем тесты. Я думаю, вы догадались оформить предыдущие варианты функции main в виде функций test_dummy, test_pointers. Напишем новую функцию test_memory:

```
void test_memory() {
    /* тестируем функции работы со списком
    // вставка элемента С выделением памяти
    struct Node * list_push_front(struct Node * list, Data d);
    struct Node * list_push_back(struct Node * list, Data d);

    // удаление 1 элемента С освобождением памяти
    Data list_pop_front(struct Node * list);
    Data list_pop_back(struct Node * list);
    Data list_delete(struct Node * t);

    // удаление всех элементов, кроме "замка"
    void list_clear(struct Node * list);
    */
    printf("----------------------- Test push/pop/delete function, use valgring now!\n");

    struct Node z;
    struct Node * list = &z;
    list_init(list);

    list_push_front(list, 5);
    list_push_front(list, 7);
    list_push_back(list, 10);
    list_push_back(list, -3);
    list_push_back(list, 22);
    list_print(list);       // 5 7 10 -3 22

    printf("popped front: %d\t", list_pop_front(list));  // 5
    list_print(list);       // 7 10 -3 22

    printf("popped back: %d\t", list_pop_back(list));  // 22
    list_print(list);       // 7 10 -3

    printf("delete node: %d\t", list_delete(list->next->next));  // 10
    list_print(list);       // 7 -3

    list_clear(list);
    printf("after clear: list_is_empty %d\n", list_is_empty(list));
}
```

Хорошо написанные тесты СОКРАЩАЮТ общее время разработки программы.

добавление данных в список

Так как в этих функциях в аргументах передаются данные типа Data, а не адреса узлов, сначала необходимо выделить под узлы память и записать в узлы данные.

```
// вставка элемента С выделением памяти
struct Node * list_push_front(struct Node * list, Data d) {
    // выделим память для нового узла t
```

```
    struct Node * t = malloc(sizeof(struct Node));
    t->data = d;                        // запишем в узел данные
    list_insert(list, t);               // вставим узел в список (с начала)
    return t;                           // вернем указатель на новый узел
}
```

Заметьте, мы используем код, написанный в функции list_insert **повторно**, а не делаем copy-paste кода. Не копируйте ошибки. Их проще найти и исправить в одном месте, чем искать по разным местам кода.

Аналогично, используя код повторно, реализуйте функцию list_push_back в **ОДНУ строку**.

удаление данных, очистка списка

Начнем с самой общей функции удаления 1 конкретного узла из списка и освобождения занимаемой узлом памяти.

Функция list_delete должна возвращать данные, которые хранятся в этом узле.

```
// удаление 1 элемента С освобождением памяти
Data list_delete(struct Node * t) {
    Data d = t->data;   // запомним данные, которые нужно возвращать
    list_remove(t);     // вытащим узел из списка
    free (t);           // освободим память (разрушим узел!)
    return d;           // вернем данные, которые хранились в разрушенном узле
}
```

Реализуйте самостоятельно в 1 строку функции list_pop_front и list_pop_back.

Реализация функции list_clear должна занимать 2 строки.

Проверьте все функции.

Запустите тесты под valgrind и убедитесь, что ваши функции работают с памятью корректно.

Код в одном файле

Для удобства посылки в проверяющую систему, все, что не нужно посылать на сервер (декларация типов и функция main) заключены в команды условной компиляции

```
#ifdef AAA
...
#endif
```

Предполагая, что макрос AAA не будет определен в проверяющей системе.

```
#include <stdio.h>
#include <stdlib.h>
#ifdef AAA
typedef int Data;       // покажем список на примере целых чисел

struct Node {           // один узел списка
    Data data;          // данные
    struct Node * prev; // указатель на предыдущий узел
    struct Node * next; // указатель на следующий узел
};

// делает список пригодным для работы
void list_init(struct Node * list);

// вставка и удаление элемента БЕЗ выделения памяти (операция с узлами)
void list_insert(struct Node * list, struct Node * t);
void list_insert2(struct Node * list, struct Node * t);
void list_insert_before(struct Node * list, struct Node * t);
void list_remove(struct Node * t);

// вставка элемента С выделением памяти
struct Node * list_push_front(struct Node * list, Data d);
struct Node * list_push_back(struct Node * list, Data d);

// удаление 1 элемента С освобождением памяти
Data list_pop_front(struct Node * list);
Data list_pop_back(struct Node * list);
Data list_delete(struct Node * t);

// удаление всех элементов, кроме "замка"
void list_clear(struct Node * list);
```

```
// вспомогательные функции печати и проверки на пустоту
void list_print (struct Node * list);
int list_is_empty(struct Node * list);

int test_pointers() {
    struct Node z;              // "замок"
    struct Node * list = &z;    // указатель на список
    struct Node a = {5}, b = {7}, c = {-3}, d = {10}, f = {22};

    printf("------------------------ Test init, print, insert/remove functions\n");
    list_init(list);
    list_print(list);       // ничего не печатается

    printf("is_empty = %d\n", list_is_empty(list));

    list_insert(list, &c);  // очень похоже на стек?
    list_insert(list, &b);
    list_insert(list, &a);
    list_print(list);       // 5 7 -3

    printf("is_empty = %d\n", list_is_empty(list));

    list_insert(list->next, &d);
    list_insert_before(list, &f);
    list_print(list);       // 5 10 7 -3 22

    list_remove(&d);
    list_remove(&f);
    list_print(list);       // 5 7 -3

    return 0;
}
void test_memory() {
    /* тестируем функции работы со списком
    // вставка элемента С выделением памяти
    struct Node * list_push_front(struct Node * list, Data d);
    struct Node * list_push_back(struct Node * list, Data d);

    // удаление 1 элемента С освобождением памяти
    Data list_pop_front(struct Node * list);
    Data list_pop_back(struct Node * list);
    Data list_delete(struct Node * t);

    // удаление всех элементов, кроме "замка"
    void list_clear(struct Node * list);
    */
    printf("------------------------ Test push/pop/delete function, use valgring now!\n");

    struct Node z;
    struct Node * list = &z;
    list_init(list);

    list_push_front(list, 5);
    list_push_front(list, 7);
    list_push_back(list, 10);
    list_push_back(list, -3);
    list_push_back(list, 22);
    list_print(list);       // 5 7 10 -3 22

    printf("popped front: %d\t", list_pop_front(list));  // 5
    list_print(list);       // 7 10 -3 22

    printf("popped back: %d\t", list_pop_back(list));  // 22
    list_print(list);       // 7 10 -3

    printf("delete node: %d\t", list_delete(list->next->next));  // 10
    list_print(list);       // 7 -3

    list_clear(list);
    printf("after clear: list_is_empty %d\n", list_is_empty(list));
}
int main() {
    test_pointers();
    test_memory();
    return 0;
```

```
}
#endif

void list_print(struct Node * list) {
    struct Node * p;
#ifdef LIST_DBG
    printf("LIST:\tthis=%p prev=%p next=%p\n", list, list->prev, list->next); // замок
#endif
    for (
        p = list->next;     // первая бусина - после замка
        p != list;          // печатаем только бусины, стоп если дошли до замка
        p = p->next
    ) {
#ifdef LIST_DBG
        printf("%d\tthis=%p prev=%p next=%p\n", p->data, p, p->prev, p->next); // печать чисел
#else
        printf("%d ", p->data); // печать чисел
#endif
    }
    printf("\n");
}

// вставляем узел t в список после узла p
void list_insert(struct Node * p, struct Node * t) {
    // объявим новую переменную n - указатель на следующий узел списка после p
    // (в заглушке это был узел b)
    struct Node * n = p->next;

    // вставим узел t после узла p
    p->next = t;        // запишем адрес 700 в поле next узла с адресом 100
    n->prev = t;        // запишем адрес 700 в поле prev узла с адресом 200
    t->next = n;        // запишем адрес 200 в поле next узла с адресом 700
    t->prev = p;        // запишем адрес 100 в поле prev узла с адресом 700
}

// вставляем узел t в список перед узлом n
void list_insert_before(struct Node * n, struct Node * t) {
    list_insert(n->prev, t);
}

// удаляем узел t из списка
void list_remove(struct Node * t) {
    struct Node * p = t->prev;
    struct Node * n = t->next;
    p->next = n;
    n->prev = p;
}

// инициализация списка: следующий и предыдущий для "замка" - сам "замок"
void list_init(struct Node * list) {
    list->next = list;
    list->prev = list;
}

// вставка элемента С выделением памяти
struct Node * list_push_front(struct Node * list, Data d) {
    // выделим память для нового узла t
    struct Node * t = malloc(sizeof(struct Node));
    t->data = d;                        // запишем в узел данные
    list_insert(list, t);               // вставим узел в список (с начала)
    return t;                           // вернем указатель на новый узел
}
struct Node * list_push_back(struct Node * list, Data d) {
    // тут нужно написать код
}

// удаление 1 элемента С освобождением памяти
Data list_delete(struct Node * t) {
    Data d = t->data;   // запомним данные, которые нужно возвращать
    list_remove(t);     // вытащим узел из списка
    free (t);           // освободим память (разрушим узел!)
    return d;           // вернем данные, которые хранились в разрушенном узле
}
Data list_pop_front(struct Node * list) {
    // тут нужно написать код
}
```

```
Data list_pop_back(struct Node * list) {
    // тут нужно написать код
}

// удаление всех элементов, кроме "замка"
void list_clear(struct Node * list) {
    // тут нужно написать код
}
int list_is_empty(struct Node * list) {
    // тут нужно написать код
}
```

Дополнительные задачи

Реализуйте функции

Функция ищет первое вхождение данных d в список, возвращает указатель на найденный узел или NULL, если данных d нет в списке.

```
struct Node * list_find(struct Node * list, Data d);
```

Функция ищет последнее вхождение данных d в список, возвращает указатель на найденный узел или NULL, если данных d нет в списке.

```
struct Node * list_find_back(struct Node * list, Data d);
```

Функция считает сколько узлов (кроме "замка") содержит список

```
int list_size(struct Node * list);
```

Функция считает, сколько раз число d входит в список list

```
int list_count(struct Node * list, Data d);
```

Функция делает обратный список, т.е если в списке были числа 1 2 3 4, то список станет содержать числа 4 3 2 1.

```
void list_revers(struct Node * list);
```

```

```

```

```

-- TatyanaDerbysheva - 11 Feb 2019