Федеральное государственное бюджетное

образовательное учреждение

высшего образования

«МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ»

(национальный исследовательский университет)

Факультет № 8 «Компьютерные науки и прикладная математика»

Кафедра 806 «Вычислительная математика и программирование»

Задание № 8
«Линейные списки»

по дисциплине «Практикум на ЭВМ»

2 семестр

Студент	Мезенин О.А.
Группа	М8О-106Б-21
Преподаватель	Дубинин А.В.
Оценка	

Москва, 2022

Содержание

1 Введение

Линейный список представляет собой динамическую упорядоченную структуру данных с однотипными элементами (узлами), которые могут повторяться. Каждый элемент связан со следующим, как правило, при помощи ссылки. Линейные списки позволяют организовывать множества переменного размера, где включение, поиск и удаление элементов могут выполняться в произвольных последовательно достигаемых местах с сохранением порядка следования остальных элементов.

Целью работы является знакомство с видами связных списков и физическим представлением связного списка. В ходе работы также необходимо написать программу для обработки связного списка.

2 Виды списков

По количеству полей различают однонаправленный (односвязный) и двунаправленный (двусвязный) списки. Каждый элемент в односвязном списке содержит один указатель на следующий элемент. В двусвязном списке элемент содержит два указателя - на следующий элемент и на предыдущий.

По способу связи различают линейные и циклические списки. В линейном списке последний элемент указывает на NULL. В циклическом списке последний элемент связан с первым.

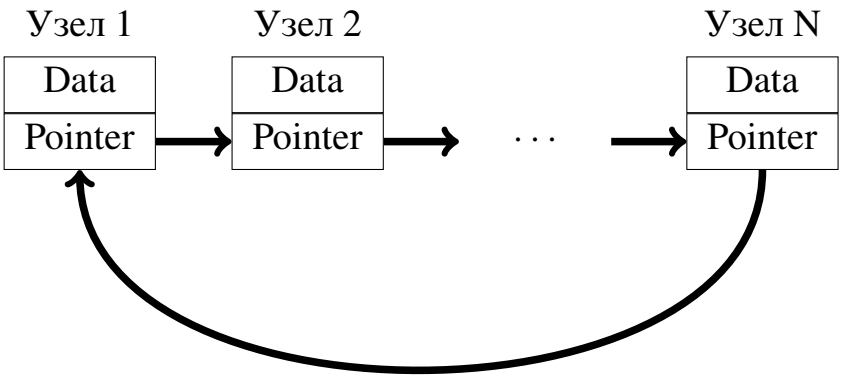
Таким образом, выделяют 4 основных вида списков: односвязный линейный список, односвязный циклический список, двусвязный линейный список, двусвязный циклический список.

2.1 Односвязный линейный список

Каждый узел в односвязном линейном списке содержит один указатель на следующий элемент, а указатель последнего узла ссылается на NULL.

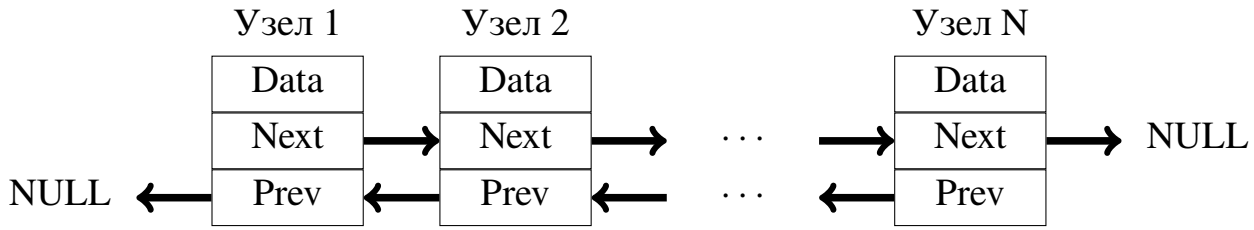
2.2 Односвязный циклический список

Каждый узел в односвязном циклическом списке содержит один указатель на следующий элемент, а указатель последнего узла ссылается на первый элемент.



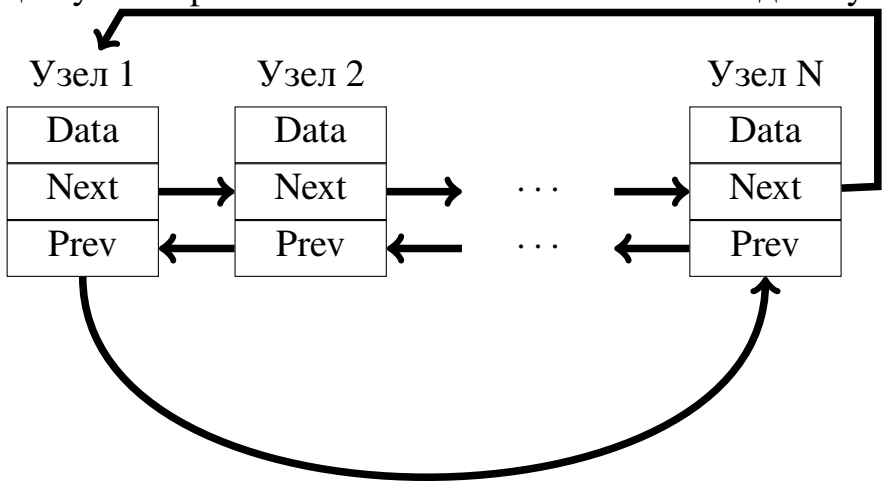
2.3 Двусвязный линейный список

Каждый узел двусвязного линейного списка содержит два поля указателей: на следующий и предыдущий узел. Поле указателя на следующий узел последнего элемента ссылается на NULL. Поле указателя на предыдущий узел первого элемента тоже ссылается на NULL.



2.4 Двусвязный циклический список

Каждый узел двусвязного циклического списка содержит два поля указателей: на следующий и предыдущий узел. Поле указателя на следующий узел последнего элемента ссылается на первый узел. Поле указателя на предыдущий узел первого элемента ссылается на последний узел.



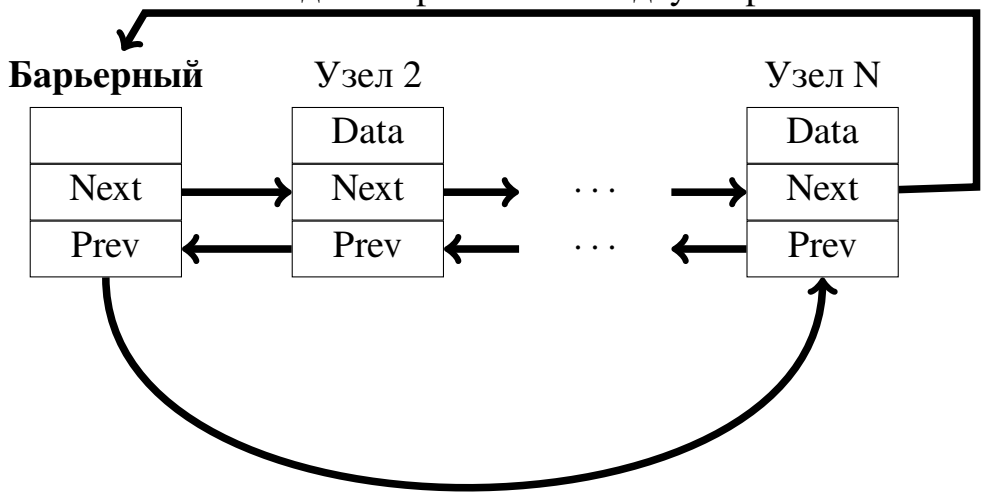
2.5 Список с барьерным элементом

В структурах рассмотренных списков и их элементов всегда могут содержаться указатели на NULL: в циклическом списке на NULL будет указывать поле указателя на первый элемент, если список пустой; а в связном списке указатель на NULL обязательно будет у поля последнего элемента.

Идея списка с барьерным элементом заключается в том, чтобы ни при каких обстоятельствах не получать нулевых указателей. Для этого в структуру

списка помещается элемент, который логически не находится в списке, но физически включён в цепочку элементов.

Список с барьерным элементом считается линейным, потому что указатель последнего элемента не ведёт на первый. Список с барьерным элементом также может быть однонаправленным и двунаправленным.



2.6 Оптимизация двунаправленного списка

В каждом элементе двунаправленного списка помимо самих данных хранятся два указателя, а в однонаправленном - один. Но можно реализовать двунаправленный список, занимающий столько же памяти, сколько однонаправленный.

Для этого используем операцию XOR, у которой есть свойство: $c = a \,\hat{}\, b \iff c \,\hat{}\, b = a$. В структуре элемента будем хранить не отдельные указатели на следующий и предыдущий элементы, а их XOR: $pn = Prev \,\hat{}\, Next$. Тогда адрес следующего (предыдущего) элемента можно получить как XOR поля pn текущего элемента и адрес предыдущего (следующего) элемента. Адрес предыдущего посещённого элемента будем хранить в дополнительном поле *итератора*.

3 Итератор и физическое представление списка

3.1 Итератор

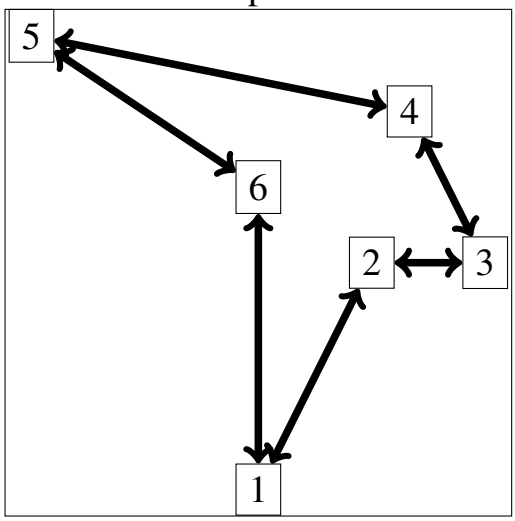
Итератор - это объект, который позволяет перемещаться по элементам некоторой последовательности и получать элементы этой последовательности. Также итератор может поддерживать изменение значения элементов, добавление и удаление элементов из последовательности.

Главным преимуществом итераторов является одинаковый интерфейс. Пользователю не нужно каждый раз подстраиваться под индивидуальный интерфейс каждой структуры данных - он может воспользоваться итератором и работать со структурой данных даже без знания принципа её устройства.

3.2 Динамические структуры

Существует два варианта физического представления связного списка: на динамических структурах и на массиве.

При физическом представлении на динамических структурах память выделяется постепенно для каждого узла, и элементы могут быть разбросаны в разных местах heap'а:

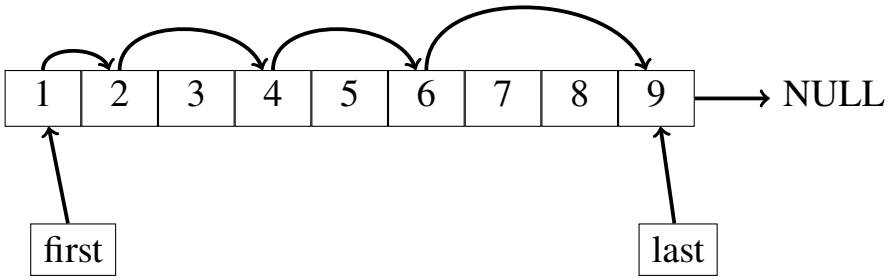


Но у такого представления есть недостаток: оно приводит к фрагментации памяти.

3.3 Массив

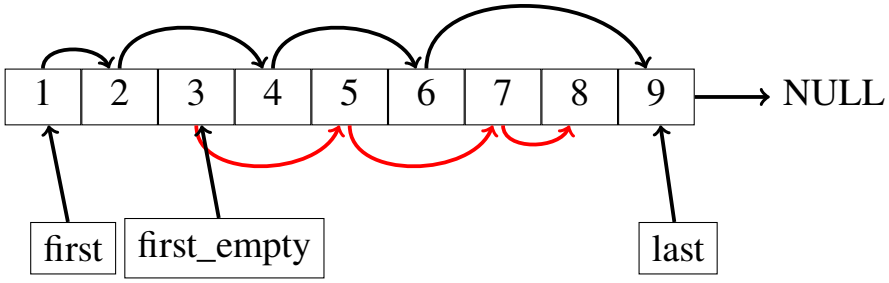
Физическое представление на массиве является более совершенным и избавляет от проблемы фрагментации памяти. В такой реализации указатель на

следующий элемент будет хранить уже не ссылку, а индекс в массиве.



Необходимо где-то хранить элементы, не входящие в список логически (свободные ячейки), либо каждый раз искать свободный элемент. В первом случае будет использована дополнительная память $O(n)$, во втором - дополнительное время на поиск элемента $O(n)$.

Но можно избавится от этих недостатков следующим образом: связать свободные элементы так же, как связываются занятые элементы, и хранить индекс первого свободного элемента в структуре списка. Тогда при добавлении нового элемента можно будет использовать первый свободный элемент, а хранимый индекс первого свободного элемента перемещать на следующий свободный элемент.



4 Практика

Задание: составить и отладить программу на языке Си для обработки линейного двунаправленного списка с барьерным элементом с отображением на массив. Элементы списка должны быть целого типа. Навигацию по списку следует реализовать с применением указателей. Предусмотреть выполнение следующих действий:

1. Печать списка.
2. Вставка нового элемента.
3. Удаление элемента из списка.
4. Подсчет длины списка.
5. Исключение из списка последних k элементов. Если в списке менее k элементов, то не менять его.

4.1 Описание структур

Воспользуемся оптимизацией двунаправленного списка, описанной в пункте 2.6. Тогда структуры элемента и листа будут выглядеть следующим образом:

```
typedef struct listel listel;
struct listel { // структура элемента
    int val;
    int pn;
};

typedef struct { // структура листа
    int pool_size;
    listel* buf;
    int barrier;
    int first;
    int size;
    int first_empty;
    int last_add;
} barrier_list;
```

Поля $first$ и $last\_add$ в структуре листа необходимо хранить, чтобы через поле pn в структуре элемента получать следующий и предыдущий элементы.

Определим структуру итератора и его функции:

```
typedef struct { // структура итератора
    barrier_list *lst;
    int prev;
    int cur;
} list_iter;

list_iter list_iter_begin(barrier_list *l);
list_iter list_iter_end(barrier_list *l);
bool list_iter_equal(list_iter it1, list_iter it2);
void list_iter_move_next(list_iter *it);
void list_iter_move_back(list_iter *it);
int list_iter_get(list_iter *it);

void list_iter_set(list_iter *it, int val);
bool list_iter_insert_before(list_iter *it, int val);
void list_iter_remove(list_iter *it);

list_iter list_iter_find(barrier_list *l, int val);
```

4.2 Реализация функций

Опишем функции получения следующего и предыдущего элементов:

```
static int get_next(list_iter *it) {
    return it->lst->buf[it->cur].pn ^ it->prev;
}

static int get_prev_before(list_iter *it) {
    return it->cur ^ it->lst->buf[it->prev].pn;
}
```

Функции инициализации листа и его удаления будут выглядеть следующим образом:

```
bool list_init(barrier_list *l) {
    l->buf = malloc((INITIAL_SIZE+1) * sizeof(listel));
    if (l->buf == NULL) return false;
    l->pool_size = INITIAL_SIZE;
    l->barrier = INITIAL_SIZE;
    l->first = l->barrier;
    l->size = 0;
    // связываем пустые элементы:
```

```
    for (int i = 1; i < l->pool_size; ++i) {
        l->buf[i].pn = (i-1)^(i+1);
    }
    l->buf[0].pn = 1^l->barrier;
    l->buf[l->barrier].pn = 0;
    l->first_empty = 0;
    l->last_add = l->barrier;
    return true;
}
void list_destroy(barrier_list *l) {
    list_iter it = list_iter_begin(l);
    while (!list_iter_equal(it, list_iter_end(l))) {
        list_iter_remove(&it);
    }
}
```

Функции вставки и удаления элементов с помощью итератора:

```
bool list_iter_insert_before(list_iter *it, int val) {
    if (it->lst->size >= it->lst->pool_size) {
        int old_barrier = it->lst->barrier;
        if (!grow_buffer(it->lst)) return false;
        if (it->cur == old_barrier) it->cur = it->lst->barrier;
        if (it->prev == old_barrier) it->prev = it->lst->barrier;
    }
    int new_id = pop_empty(it->lst);
    listel* new_el = &it->lst->buf[new_id];
    new_el->pn = it->cur ^ it->prev;
    new_el->val = val;
    int next = get_next(it);
    if (list_get_size(it->lst) > 0) {
        it->lst->buf[it->prev].pn = get_prev_before(it) ^ new_id;
        it->lst->buf[it->cur].pn = new_id ^ next;
    }
    it->prev = new_id;
    it->lst->size++;
    if (it->cur == it->lst->first) {
        it->lst->first = new_id;
    }
    return true;
}

void list_iter_remove(list_iter *it) {
```

```
    int next = get_next(it);
    if (list_get_size(it->lst) > 1) {
        it->lst->buf[it->prev].pn = get_prev_before(it) ^ next;
        it->lst->buf[next].pn = it->lst->buf[next].pn ^ it->cur ^ it->prev;
    }
    if (it->lst->first == it->cur) {
        it->lst->first = next;
    }
    if (it->lst->first_empty != it->lst->barrier)
        it->lst->buf[it->lst->first_empty].pn =
                it->lst->buf[it->lst->first_empty].pn ^
                it->lst->last_add ^ it->cur;
    it->lst->buf[it->cur].pn = it->lst->last_add ^ it->lst->first_empty;
    it->lst->first_empty = it->cur;
    it->cur = next;
    it->lst->size--;
}
```

Вставка и удаление элемента работают за $O(1)$. При вставке итератор сдвигается на новый элемент, при удалении - на следующий.

Удаление элементов будет осуществляться по значению. Для этого нам нужна функция поиска элемента, которая будет возвращать итератор на первый найденный элемент с нужным значением. Если элемент не найден, возвращает итератор, указывающий на последний элемент.

```
list_iter list_iter_find(barrier_list *l, int val) {
    list_iter iter = list_iter_begin(l);
    for (;
    !list_iter_equal(iter, list_iter_end(l));
    list_iter_move_next(&iter)
    ) {
        if (list_iter_get(&iter) == val) break;
    }
    return iter;
}
```

Поиск элемента работает за $O(n)$.

Функции для работы с пользователем вынесем в отдельные файлы user_interface.h и user_interface.c.

Функция вывода элементов списка работает за $O(n)$:

```
void print_list(barrier_list *lst) {
    if (list_get_size(lst) == 0) {
        printf("list is empty\n");
        return;
    }
    list_iter it = list_iter_begin(lst);
    while(!list_iter_equal(it, list_iter_end(lst))) {
        printf("%d ", list_iter_get(&it));
        list_iter_move_next(&it);
    }
    printf("\n");
}
```

Функции добавления и удаления элементов могут работать сразу с несколькими элементами. Для этого в качестве параметра этих функций указывается ссылка на вектор $svector\_int$, содержащий значения, которые нужно добавить или удалить:

```
void add(barrier_list *l, svector_int *buf) {
    list_iter iter = list_iter_begin(l);
    for (int i = 0; i < svint_get_size(buf); ++i) {
        list_iter_insert_before(&iter, svint_get(buf, i));
    }
}

void del(barrier_list *l, svector_int* buf) {
    if (list_get_size(l) == 0) {
        printf("list is empty\n");
        return;
    }
    svector_int buf_not;
    svint_init(&buf_not);
    list_iter iter;
    for (int i = 0; i < svint_get_size(buf); ++i) {
        iter = list_iter_find(l, svint_get(buf, i));
        if (list_iter_equal(iter, list_iter_end(l)))
            svint_push_back(&buf_not, svint_get(buf, i));
        else
            list_iter_remove(&iter);
    }
    if (svint_get_size(&buf_not) != 0) {
        printf("next elements not in the list: ");
        int buf_not_size = svint_get_size(&buf_not);
```

```
        for (int i = 0; i < buf_not_size; ++i) {
            printf("%d ", svint_pop_front(&buf_not));
        }
        printf("\n");
    }
}
```

Пусть v - количество входных значений, n - размер списка. Функция добавления работает за $O(v)$. Функция удаления работает за $O(v * n)$.

Функция *pop* удаляет последние K элементов. Работает за $O(K)$.

```
void pop(barrier_list *l, int k) {
    if (list_get_size(l) < k) {
        printf("list size less than %d\n", k);
        return;
    }
    list_iter iter = list_iter_end(l);
    list_iter_move_back(&iter);
    for (int i = 0; i < k; ++i) {
        list_iter_remove(&iter);
        list_iter_move_back(&iter);
    }
}
```

4.3 Тесты

```
h, help - output help
add VALUE [VALUE2]... - add VALUE(s) to list (values must be integers)
show - print list
del VALUE [VALUE2]... - delete VALUE(s) from list (if there are several
elements having the same value, the first one will be deleted)
pop K - delete last K elements
exit - exit program
show
list is empty
add 1 2 3
show
1 2 3
add 5 6 7 8 9
show
5 6 7 8 9 1 2 3
```

```
del 1 3 2
show
5 6 7 8 9
add dad 2
wrong command
skdl add 3
wrong command
add 1 -2- 4-
wrong command
add 1 -2 -0
show
1 -2 0 5 6 7 8 9
pop 3
show
1 -2 0 5 6
pop 10
list size less than 10
add 1 1
del 1 9
next elements not in the list: 9
show
1 1 -2 0 5 6
exit
```

5 Выводы

В ходе работы были рассмотрены виды связных списков, возможность оптимизации двусвязного списка, итератор и варианты физического представления списка. Список представляет собой структуру данных, в которой вставка и удаление работают за $O(1)$ и могут выполняться в любом месте последовательности данных, поиск элемента же работает за $O(n)$.

6 Список источников

1. Связные списки - https://prog-cpp.ru/data-list/
2. Что такое итераторы и зачем они нужны - https://ru.stackoverflow.com/questions/270697/Что-такое-итераторы-и-зачем-они-нужны