Московский авиационный институт
(Национальный Исследовательский Институт)

Кафедра вычислительной математики и программирования.

Курсовая работа

по курсу «практикум на ЭВМ»

Задание №8:

Линейные списки

Студент:	Суханов Е. А.
Группа:	М80-106Б
Преподаватель:	Дубинин А. В.
Оценка:	
Дата:	

Москва
2020

Оглавление

Введение

Цель. Научиться применять списки и итераторы на практике.

Задание. Составить и отладить программу на языке Си для обработки однонаправленного связного линейного списка на динамическом массиве. С применением итераторов для навигации. Программа должна выполнять следующие действия в интерактивном режиме:

- Печать списка;
- Вставка нового элемента в список;
- Удаление элемента из списка;
- Подсчет длины списка;
- Переставить элементы списка в обратном порядке;

Теория

Линейный список. Списком называется упорядоченная совокупность объектов одной природы (одного типа), при этом, в отличии от массива, порядок элементов определяется не индексами, а расположением относительно других элементов списка. Примером списка может служить очередь людей в магазине или в больнице: расположение человека определяется относительного его соседей по очереди. Еще одним примером является железнодорожный подвижной состав: К локомотиву цепляется вагон, а к этому вагону цепляют еще один, и т.д.. И все же, эти примеры не полностью раскрывают структуру списка. Поэтому займемся более формальным определением списка.

Существуют две вариации линейного списка:

- Односвязный список, элементы которого связаны только в одну сторону (имеют одно отношение порядка);

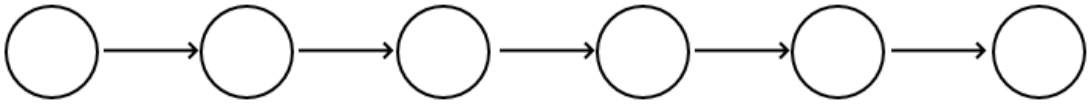

Рис. 1 Односвязный список

- Двусвязные список, элементы которого связаны в две стороны (имеют два отношения порядка);

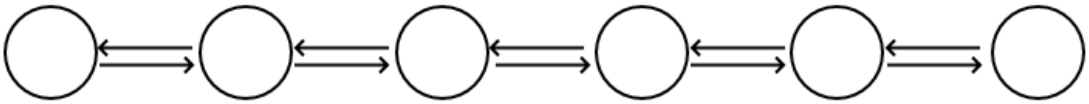

Рис. 2: Двусвязный список

То есть в двусвязном списке можно переходить к следующему и предыдущему элементу, а в односвязном – только к следующему.

Функциональная спецификация двусвязного списка

- СОЗДАТЬ: Создает пустой список L_T; Сложность O(1);
- ПУСТО: Проверяет пуст ли список L_T; Сложность O(1);

- ДЛИНА: Возвращает кол-во элементов списка L_T, Сложность O(1);
- ПЕРВЫЙ: Возвращает первый элемент списка L_T, Сложность O(1);
- ПОСЛЕДНИЙ: Возвращает последний элемент списка L_T, Сложность O(1);
- СЛЕДУЮЩИЙ: Возвращает следующий за данным элементом элемент списка L_T, Сложность O(1);
- ПРЕДЫДУЩИЙ: Возвращает предыдущий за данным элементом элемент списка L_T, Сложность O(1);
- ВСТАВКА: Вставляет новый элемент пред каким-то элементом списка L_T, Сложность O(1);
- УДАЛЕНИЕ: Удаляет какой-то элемент списка L_T; Сложность O(1);
- УНИЧТОЖИТЬ: Уничтожает список L_T, Сложность O(n);

Нужно обратить внимание, что, в отличие от массива, вставка элемента имеет сложность O(1), а не O(n). Однако, другая сторона относительной упорядоченности, заключается в линейном времени обращении к элементу списка.

Логическое описание. Линейный список можно реализовать тремя способами:

- На динамическом массиве (Или сразу на векторе);
- Цепочка динамических элементов;
- На файле;

Рис. 3: Элемент односвязного списка

Все эти способы имеют общую вещь: идейно одинаковое устройство элемента списка. На рис. 3 изображен элемент односвязного списка, зеленным отмечено поле, хранящее значение элемента, а оранжевым – поле, хранящее указатель/индекс следующего элемента. Элемент двусвязного списка отличается лишь указателем/индексом на предыдущий элемент).

Цепочка динамических элементов. Самая простая реализация списка. Идея заключается в выделении памяти под каждый элемент списка, а затем их связывании с помощью указателей. Сам список будет иметь следующую структуру:

```
typedef struct list_el list_el;
typedef struct list list;
struct list_el{
    T val;              // Значение элемента списка
    list_el *next;      // Указатель на следующий элемент
};

struct list{
    int size;           // Хранит кол-во элементов списка
    lsit_el *first;     // Указатель на первый элемент списка
    list_el *last;      // Указатель на последний элемент списка
};
```

Приведем пример функции вставки элемента:

```
// list_append – вставляет новый элемент со значением val после
curr_el
bool list_append(list* l, list_el* curr_el, T val){
     // Выделяем память под новый элемент списка
     list_el* new_el = malloc(sizeof(list_el));

     // Если произошла ошибка выделения памяти
     if(new_el == NULL){
          return false;
     }

     new_el->val  = val;

     // Меняем соответствующие указатели
     new_el->next = curr_el->next;
     curr_el->next = new_el;

     l->size++;
     return true;
}
```

Вставка нового элемента, а так же удаление сводится к изменению соответствующих связей. Стоит обратить внимание на освобождение памяти, которую занимают элементы списка.

Необходимость хранения указателя на последний элемент, а так же размера списка нужна для соблюдения функциональной спецификации (указатель на последний элемент списка позволяет получить его за O(1), а переменная size позволяет за O(1) узнать размер списка, а так же пуст ли он). Кроме того, указатель на последний элемент позволяет добавлять новые элементы в конец за O(1) (не нужно искать указатель на последний элемент, он всегда под рукой).

Список на динамическом массиве. Рассмотрим реализацию списка на динамическом массиве.

Важно понять, что основное отличие списка на динамическом массиве от списка в виде цепочки элементов заключается в выделении памяти под его элементы. Во-первых, указатели на элементы заменяются их индексами. А во-вторых реализация выделения памяти под элемент и ее освобождения (malloc и free) заменяются на аналогичные функции, работающие в рамках динамического массива. Другими словами, если раньше в качестве массива мы использовали оперативную память, то теперь мы используем массив. Рассмотрим эту реализацию подробнее.

```
typedef struct list_el list_el;
typedef struct list list;

struct list_el{
     T val;         // Значение элемента списка
     size_t next;   // Индекс следующего элемента
};

struct list {
     list_el *buf;  // Указатель на динамический массив
     size_t cap;    // Размер массива
     size_t size;   // Размер списка
     size_t first;  // Индекс первого элемента списка
     size_t last;   // Индекс последнего элемента списка
     size_t empty;  // Индекс на первый пустой элемент (см. ниже)
};
```

Замечание. Для упрощения работы с динамическим массивом можно использовать структуру данных «вектор».

Как видно из листинга, мы поменяли указатели на индексы. Осталось придумать, каким образом выдавать индексы новым элементам.

Тривиальный вариант. Для нового элемента можно всегда брать некоторый, точно не занятый элемент массива, например последний. То есть, можно завести счетчик, который будет только увеличиваться, а его значение и будет индексом нового элемента. А удаление старого элемента никак не производить (выкидывать этот элемент из списка, но оставлять его в массиве).
Реализация этого варианта будет выглядеть таким образом:

```
// get_new_idx – выделяет индекс для нового элемента списка
size_t get_new_idx(list* l){
     if(l->empty == l->cap){
          // Нужно увеличить размер динамического массива
     }

     size_t new_idx = l->empty;
     l->buf[new_idx]->next = NOT_AN_INDEX;
     l->empty++;
     return new_idx;
}

// free_idx – осовобождает индекс элемента
void free_idx(list* l, size_t idx){
     return;
}
```

Например, у нас уже есть список из 4 ех элементов (см. рис. 4).

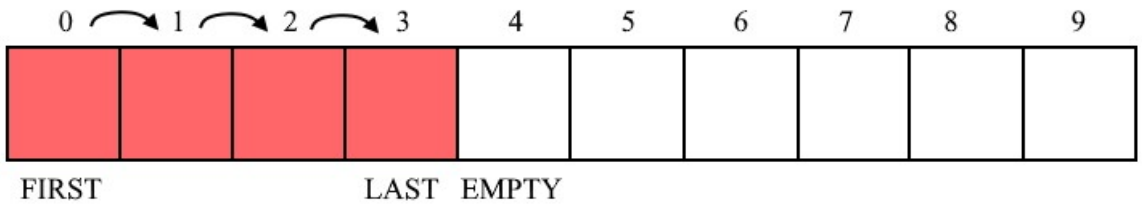

Рисунок 4: список из 4-х элементов на массиве

И мы хотим вставить новый элемент в начало списка. Нам нужно выделить индекс для нового элемента: `size_t new_el_idx = get_new_idx(l).` Таким образом переменная `new_el_idx` будет иметь значение 4, а EMPTY станет равным 5. На рис. 5 изображен список после изменений.

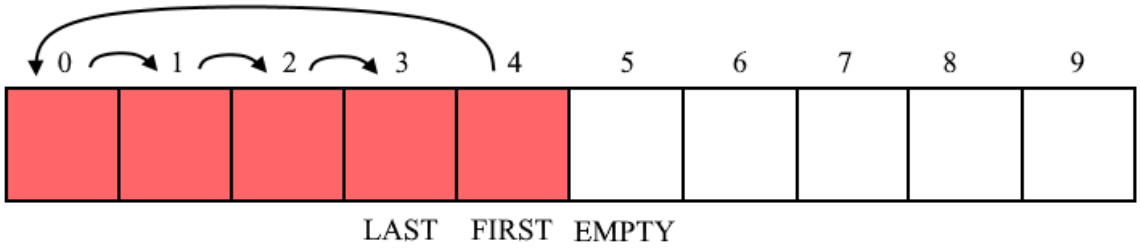

Рис. 5: результат добавления элемента в начало списка

Очевидным недостатком тривиального варианта выделения индексов является неэффективное использование памяти, так как индексы удаленных элементов не будут использоваться. Например, если мы удалим третий элемент списка (Это элемент с индексом 1), то массив будет выглядеть так: (см. рис. 6)
Значение EMPTY не изменилось, и ячейка массива с индексом 1 никогда не будет использоваться.

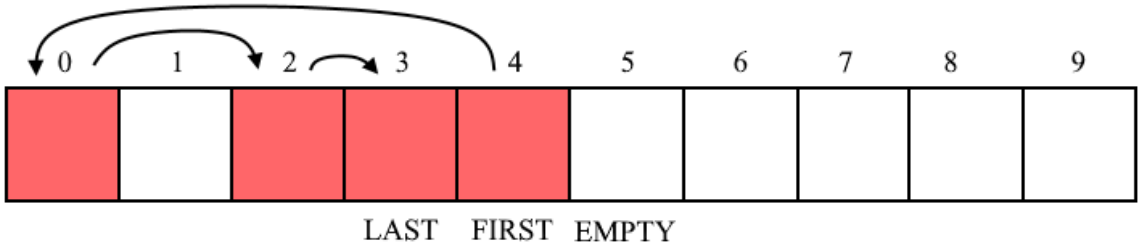

Рис. 6: результат удаления элемента из списка

Рассмотрим другой вариант, у которого нет этого недостатка.

Список пустых ячеек. Идея заключается в формировании списка пустых ячеек (если быть более точным, то стека). Тогда, если какой-то элемент массива освободится, то его достаточно будет добавить в список пустых ячеек. А если список пустых ячеек окажется пустым, то достаточно выделить еще немного памяти под массив, и обновить список пустых ячеек. Этот список не нужно хранить отдельно, мы можем использовать тот же самый массив для него.
Реализация этого варианта будет выглядеть таким образом:

```
// get_new_idx – выделяет индекс для нового элемента списка
size_t get_new_idx(list* l){
     // Если список пустых элементов пуст
     if(l->empty == NOT_AN_INDEX){
          // Нужно увеличить размер динамического массива
          // И обновить список
     }
     size_t new_idx = l->empty;
```

```
     // Обновляем empty, получаем следующий пустой элемент из
     // списка пустых элементов.
     l->empty = l->buf[l->empty]->next;

     l->buf[new_idx]->next = NOT_AN_INDEX;
     return new_idx;
}


// free_idx – освобождает индекс элемента
void free_idx(list* l, size_t idx){
     // Добавляем в начало списка пустых элементов
     l->buf[idx]->next = l->empty;
     l->empty = idx;
}
```

Теперь когда-то освобожденные индексы могут снова использоваться. Этот вариант изображен на рис. 7.

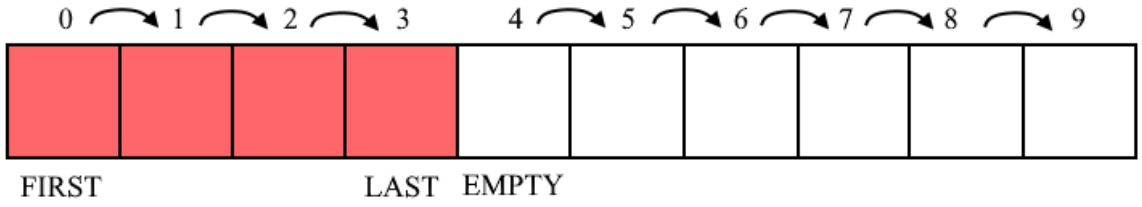

Рис. 7: исходный массив

После добавления нового элемента в начало списка (см. рис. 8).

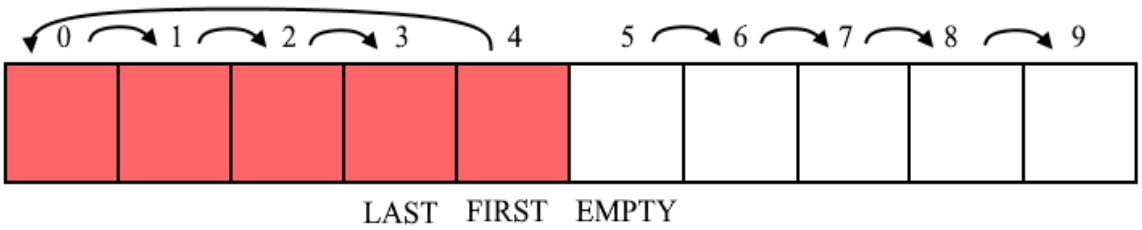

Рис. 8: Массив, после вставки в начало нового элемента

После удаления третьего элемента списка. Для удобства новая связь обозначена зеленым цветом (см. рис. 9).

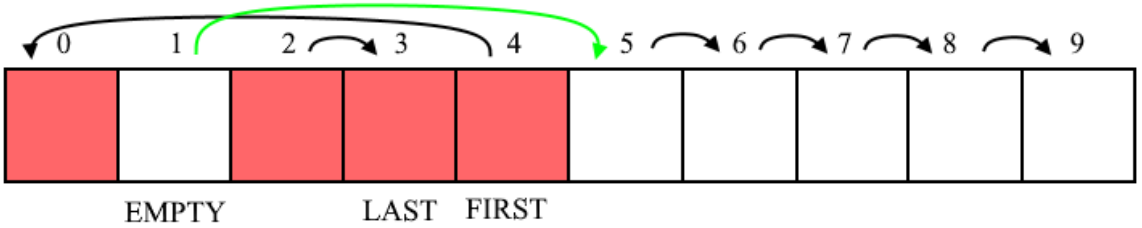

Рис. 9: Массив после удаления третьего элемента списка

Положительной стороной списка на цепочке динамических объектов является простота его реализации. Однако, из-за большого количества мелких объектов, оперативная память приложения фрагментируется, что в конечном итоге может привести к невозможности выделения большого блока данных. Список на динамическом массиве лишен этого недостатка.

Итераторы. Итераторы нужны для упрощения навигации по различным структурам данных. Они представляют одинаковый (или не очень одинаковый) интерфейс для общения с разными структурами данных. Это один из инструментов, позволяющий скрыть внутреннее устройство структуры данных.

Например итератор может содержать следующие функции:

- СЛЕДУЮЩИЙ: Принимает указатель на итератор I, и перемещает его на следующий элемент
- ПОЛУЧИТЬ: Возвращает элемент, на который указывает итератор I
- ЗАПИСАТЬ: Записывает значение в элемент, на который указывает итератор I
- РАВЕН: Сравнивает два итератора
- ВСТАВИТЬ: Вставляет новый элемент перед элементом списка, на который указывает итератор I
- УДАЛИТЬ: Удаляет элемент, на который указывает итератор

А сам список должен уметь создавать итераторы:

- НАЧАЛО: Возвращает итератор, указывающий на начало списка
- КОНЕЦ: Возвращает итератор, указывающий на конец списка

При этом КОНЕЦ должен возвращать итератор, указывающий на элемент, который находится за последним. Это нужно для упрощении использования итераторов в циклах.

Итератор – это коротко живущая сущность. И старый итератор может стать недействительным, если производить какие-то операции с его структурой данных. Поэтому, во время использования итератора следует использовать

только его функционал. Например ее можно использовать, если требуется пройтись по всему списку.

Устройство итератора односвязного списка.

Итератор может иметь следующую структуру

```
typedef struct list_it list_it;
struct list_it {
    list* l;            // Указатель на список
    size_t prev_el;     // Индекс/указатель на предыдущий элемент
};
```

Указатель на список нужен для использования функций этого списка, а индекс на предыдущий элемент нужен для возможности изменения связей элемента на который «указывает» итератор (так как в односвязном списке нельзя за константное время обратиться к предыдущему элементу, что бы изменить его связь).

Замечание. На самом деле, для работы итератора не нужен «указатель» на предыдущий элемент, нужен только указатель на поле next.

Описание программы

Разобьём задание на задачи

- **Список на динамическом массиве и итератор.** Для выполнения задания нужно реализовать структуру данных «список». Опишем функции, которые нужны для работы с ним:

bool list_init Готовит список к работе;

void list_deinit Освобождает память, выделенную под список; Уничтожает сам список;

bool list_is_empty Если список пуст, то возвращает true, иначе false;

size_t list_get_size Возвращает количество элементов в списке ;

list_iterator list_begin Возвращает итератор, указывающий на начало;

list_iterator list_end Возвращает итератор, указывающий на конец;

T list_iterator_get_val Возвращает значение элемента, на который указывает итератор;

void list_iterator_set_val
Присваивает элементу указанное значение;

void list_iterator_next Перемещает указатель на следующий элемент;

bool list_iterator_is_equals
Сравнивает два указателя, если они совпадают, то возвращает true, иначе false;

bool list_iterator_insert_before
Создает элемент с заданным значением перед элементом, на который указывает итератор. Возвращает true, если операция прошла успешно, иначе false;

bool list_iterator_remove
Удаляет элемент, на который указывает итератор, итератор перемещается вперед Возвращает true, если операция прошла успешно, иначе false;

- **Обработка команд.** Данная программа является интерактивной, поэтому необходимо реализовать считывание и обработку команд. Обработка команд происходит в два этапа.
  - **Считывание.** С потока ввода считывается команда и ее информация записывается в структуру данных «command». Этим занимается функция ***get_command***;
  - **Выполнение.** Для команды вызывается ее обработчик. Этим занимается функция ***handle_command***;
  - **Список команд.** Полный список команд можно увидеть на 278-до конца в файле command.c, а их описание в файле help.c;
- **Разворот элементов списка.** Идея алгоритма разворота:
  Имеем два итератора l – указывающий на начало списка и r – указывающий на конец.
  1. Удаляем элемент, на который указывает l, и сохраняем его значение;
  2. Вставляем его перед r, при этом сохраняя указатель r (то есть указатель r не должен переместиться);
  3. Повторяем 1 и 2 шаг пока l.next != r ;

Структура проекта. Программа разбита на модули:

- command – модуль, реализующий обработку команд. Имеется возможность простого добавления новой команды;
- help – модуль, выводящий справку о программе;
- list – реализация «шаблонного»списка на массиве (препроцессор генерирует код для определенного типа);
- log – модуль, помогающий выводить отладочную информацию;
- template – содержит макросы, которые позволяют генерировать код для разных типов;
- tests – содержит файлы для тестирования некоторых модулей программы;
- main.c – содержит точку входа в программу. А так же главный цикл;
- test.c – вызывает тесты из модуля tests;

Для упрощения сборки проекта используется утилита make. Для сборки проекта нужно запустить цель main (make main или просто make), для сборки тестирующего приложения нужно запустить цель test (make test).

Заключение

Структура данных «линейный список» имеет множество применений, поэтому с ней нужно уметь работать. А итераторы упрощают использование структур данных.

Список источников

1. Гайсарян С. С., Зайцев В. Е. Курс информатики: Учеб. Пособие. – М.: Изд-во Вузовская книга, 2012. – 424с.: ил.
2. Практикум по циклу дисциплин “Информатика”. Ч. II. 2012/13 уч. года