МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ

Федеральное государственное бюджетное образовательное учреждение

высшего профессионально образования

“Московский авиационный институт

(национальный исследовательский университет)”

**ФАКУЛЬТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И ПРИКЛАДНОЙ МАТЕМАТИКИ**

**Кафедра вычислительной математики и программирования**

Курсовой проект по курсу:

“Основы информатики”

II семестр

Задание 8

“Линейные списки на языке Си”

Работу выполнил:

студент 1 курса

Группы М8О-107Б-2018

Гамов Павел Антонович

Проверила:

асп. каф. 806 Ридли Александра Николаевна.

Научный руководитель \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Москва, 2019

**Cодержание**

Постановка задачи………………………………………………………………..3

list.h………………………………………………………………………………..3

list.c………………………………………………………………………………..4

main.c……………………………………………………………………………..12

makefile…………………………………………………………………………...13

Пример работы……………………..……………………………………………14

Заключение………………………………………………………………………15

Список использованной литературы…………………………………………..16

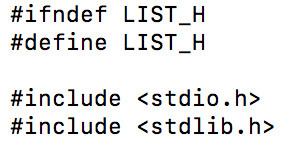
**Постановка задачи**

Составить и отладить программу на языке Си для обработки линейного списка заданной организации с отображением на динамические структуры.

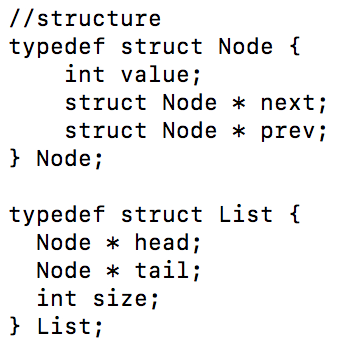
Вид списка: линейный двунаправленный, целочисленный.

Реализовать весь спектр функций для работы с данным списком, а также реализовать особенную функцию, в моем случае она звучит так: произвести вставку копий введенного значения до размера k, если в списке уже присутствует k таких элементов, не менять его.

**list.h**

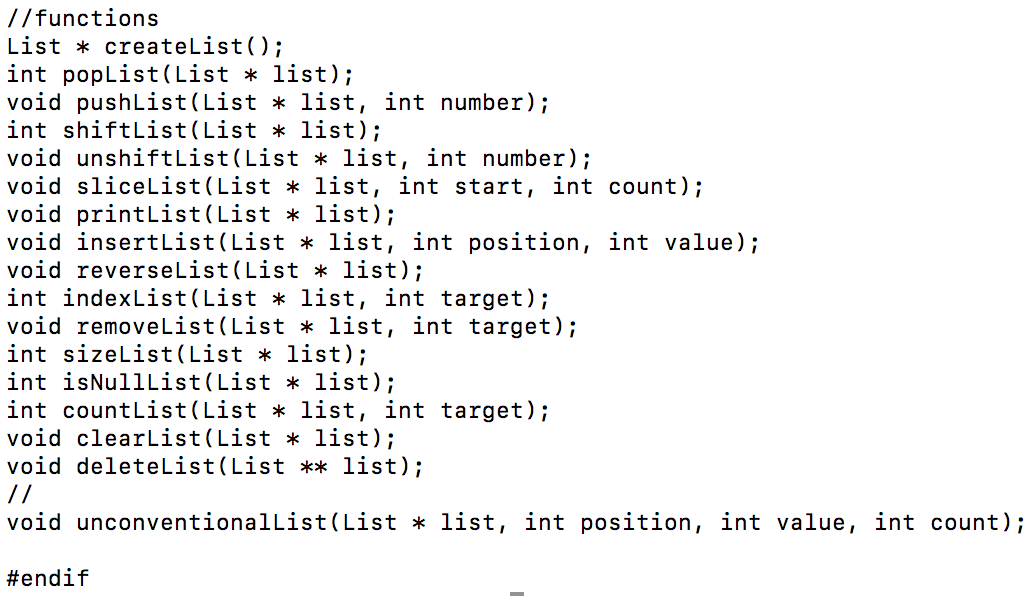


Подключаем стандартные библиотеки.



Каждый наш список будет состоять из трех вещей, ссылки на начало списка head, ссылку на конец списка tail и размер этого списка size.

Выше определяем структуру для каждого node, то есть элемента нашего списка, каждый элемент будет иметь ссылку на предыдущий, ссылку на следующий элемент, а также само целочисленное значение.

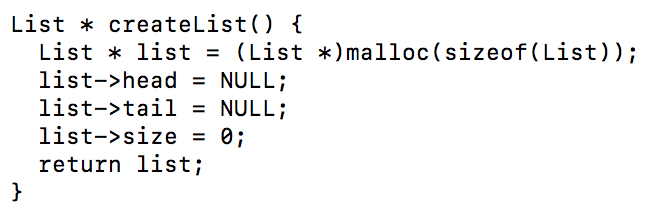


Замечаем, что функций предостаточно, так что обзор будет продолжительным.

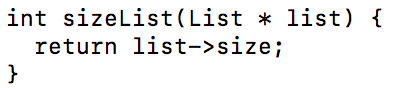
**list.c**

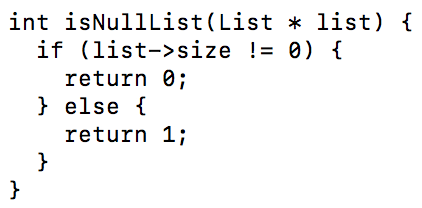


Начнем по порядку.

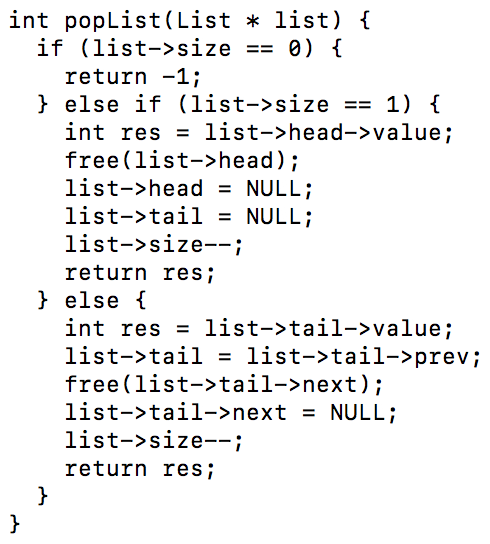


createList выделяет память под структуру, обнуляет размер и ставит пустые ссылки, возвращает ссылку на память.

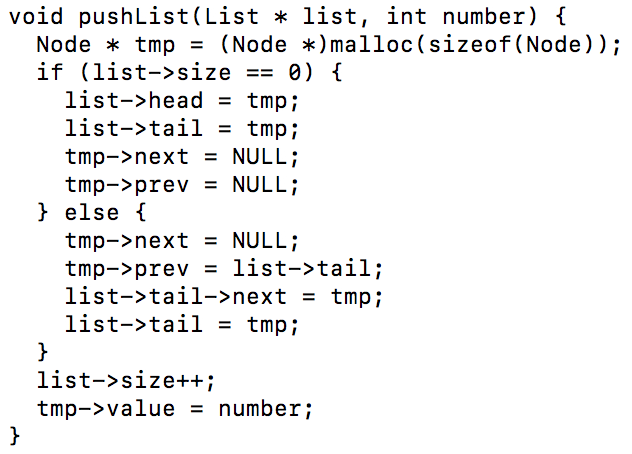


sizeList необходима когда надо в цикле пройтись по листу, возвращает размер листа в интах.  
  


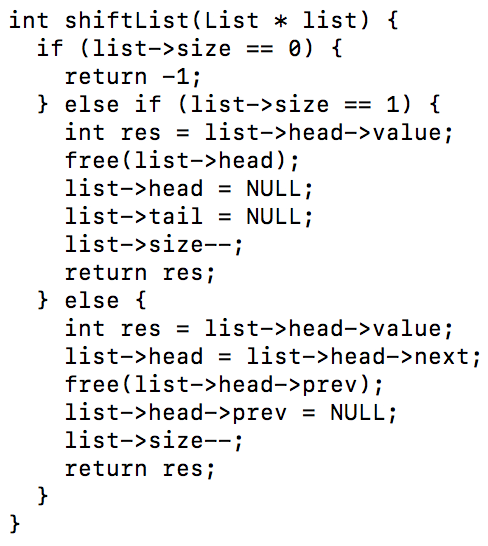
isNullList возращает ноль если список не пустой, 1 иначе.

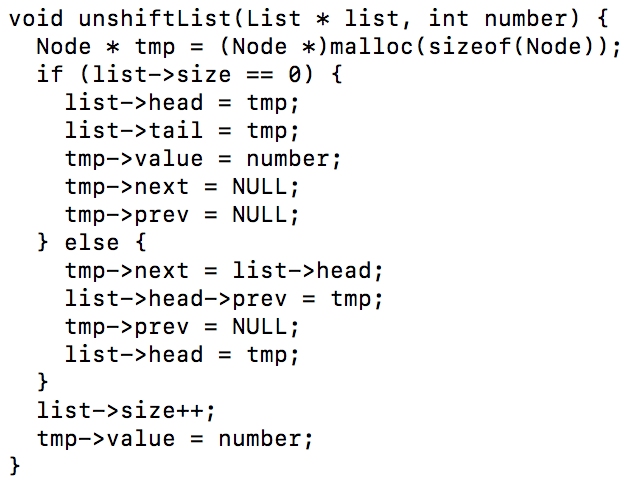


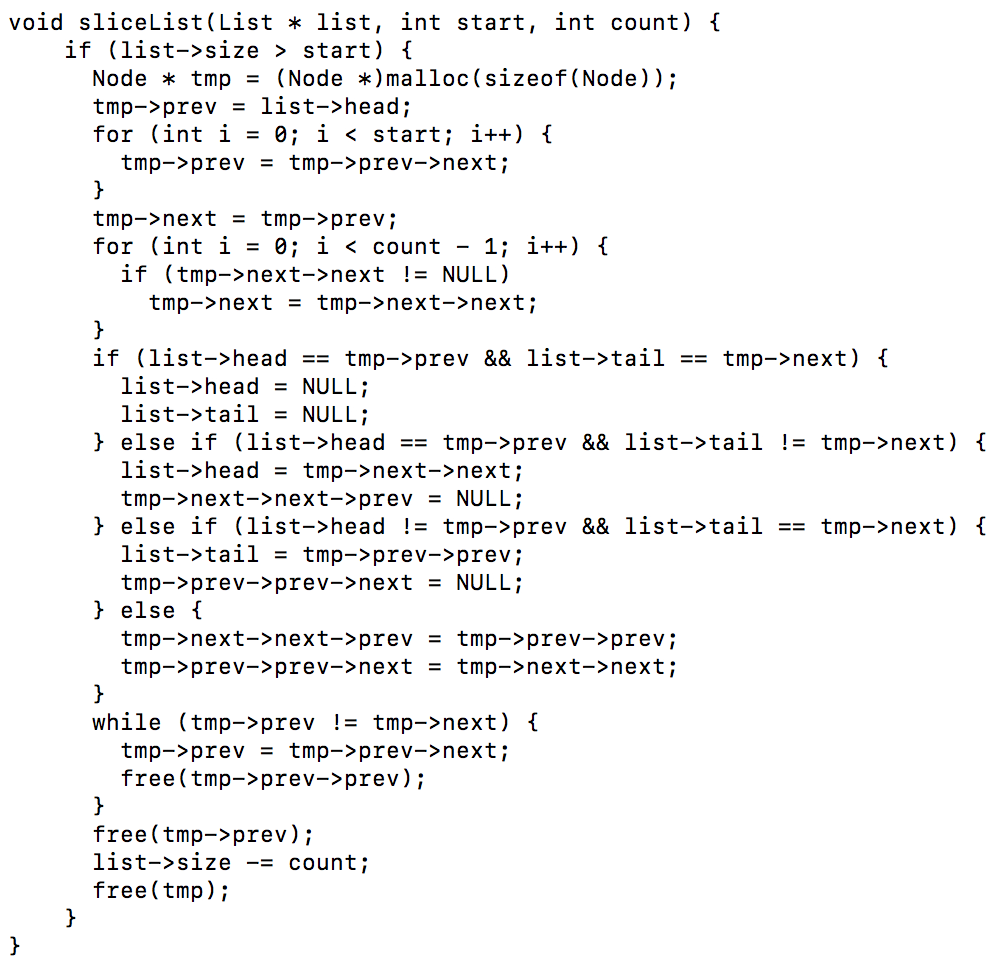
popList возвращает последний элемент если таковой вообще есть, иначе -1. При размерах 1 обнуляет ссылки head и tail, освобождает память. При больших, идет в tail и играет оттуда. Пока легкая разминка с указателями.



pushList записывает в конец списка новый элемент с указанным значением. Отдельная обработка для размера 0 и для больших. Инкрементируем размер листа.



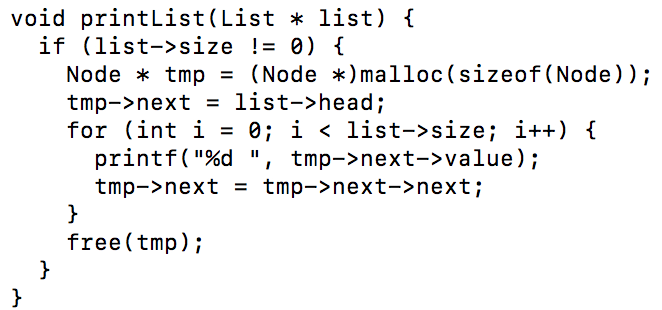
shiftList похож на popList, только возвращает значение первого элемента в списке. Похожий код.  
  


unshiftList очевидно добавляет node в начало списка, как и в случае pushList, имеем моментальный доступ к head, так что данная операция выполняется очень быстро.  
  


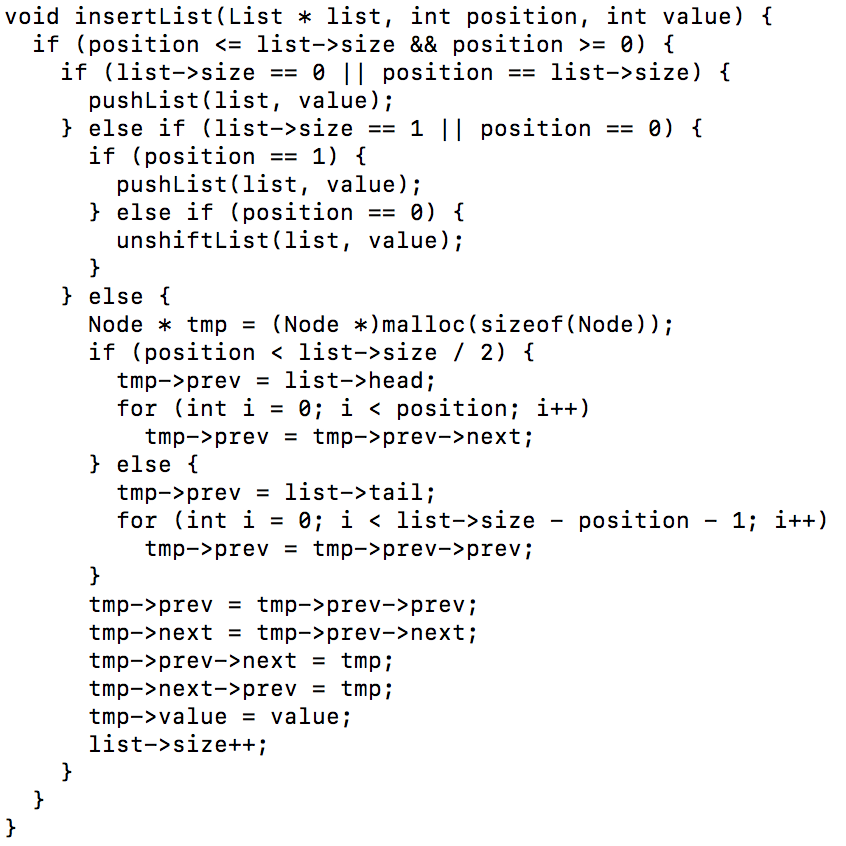
sliceList прямиком из Питона, интересная и полезная функция, на вход принимает два значения, откуда и сколько, допустим имеем 5 значений в листе, slice(2,2) идет к 2 по индексу элементу (нумерация начинается от 0) и удаляет начиная с него 2 элемента.

Из листа 1 2 3 4 5 она сделает 1 2 5.

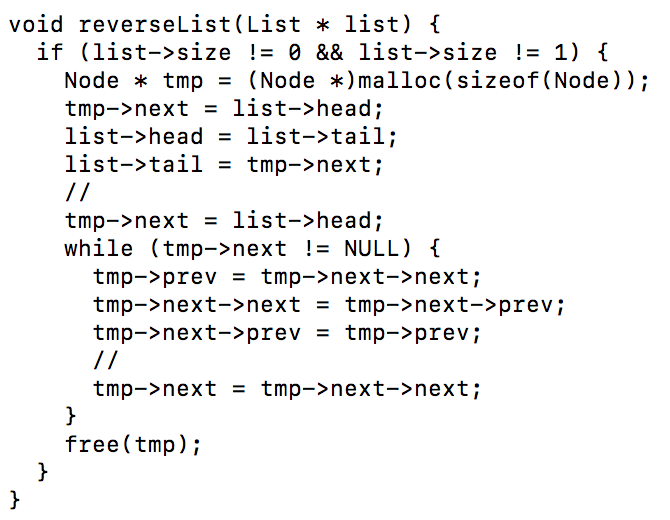
Создаем node tmp, который будет садиться на шпагат, что происходит, сейчас объясню, tmp->prev, левой ногой мы должны в цикле добраться до элемента, с которого мы будем начинать вырезать. Когда добрались, правая нога, tmp->next, начинает отдаляться от левой, добираясь до элемента до которого включительно будет произведена операция удаления. Когда tmp принял свое положения, зная границы, мы можем соединить левые и правые элементы соответственно. Отдельная обработка для ситуаций, когда обе ноги стоят на краях листа, на одном из них, или когда стоят внутри листа. После этапа соединения оставшихся элементов обратно в целый список и уменьшения размера первого получаем вырезанный кусок, который держит под собой tmp. В цикле начинаем двигать tmp->prev к tmp->next, попутно освобождая каждый node. Удалив каждый, удаляем tmp.



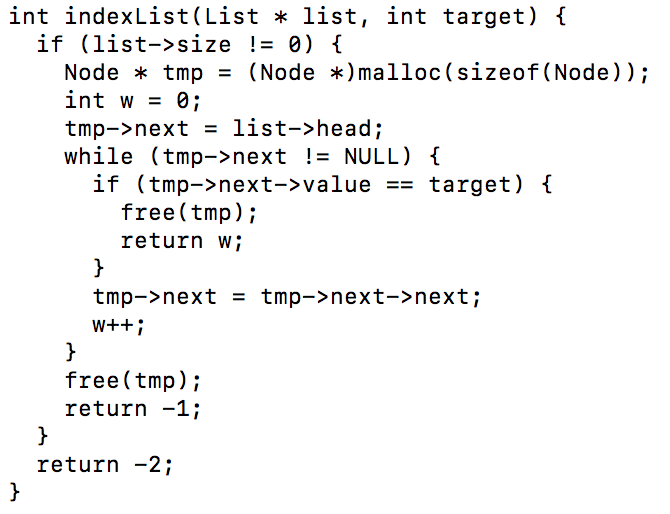
printList можно было и не помещать ее в библиотеку. Позволяет посмотреть содержимое листа. Создает tmp, которая шагает по элементам, печатая их значение, после чего удаляется.

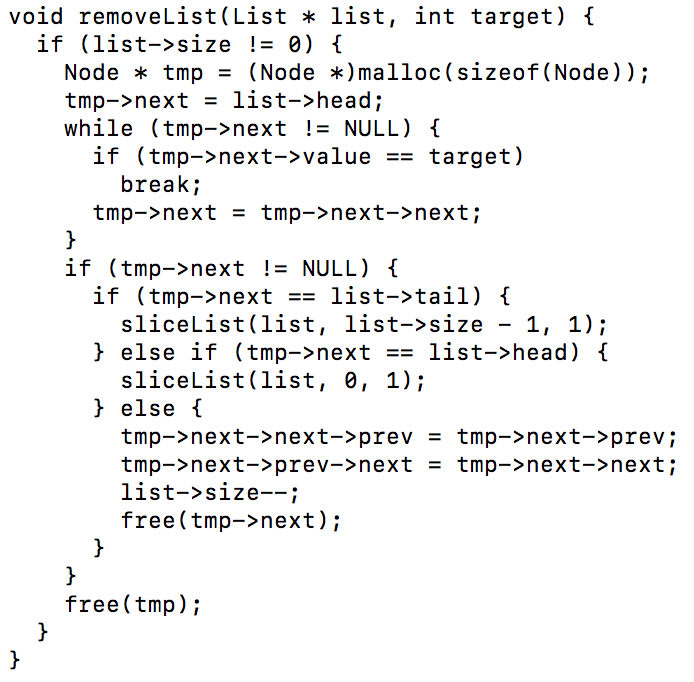


insertList вставляет в позицию значение. если лист пустой или индекс равняется краям, использует push или unshift, если надо вставить куда то в центр, выделяем tmp, шагаем слева или справа, смотря откуда легче, что обеспечивает не больше size/2 шагов, после чего обменивается ссылками с соседними элементами, вставая между ними.

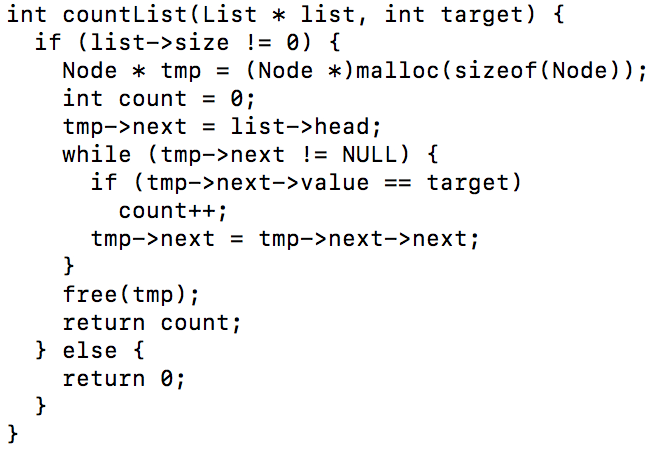


reverseList переворачивает список. Выделяем tmp, шагаем меняя prev на next и наоборот. Меняем местами head и tail. Освобождаем tmp.

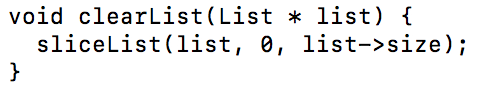


indexList она же включает в себя функцию isIn из питона вроде, принимает цель, значение которое нужно искать, бежит от начала, возвращает индекс первого совпадения цели, иначе -1 если данного элемента нет в списке, -2 если список пустой.  
  


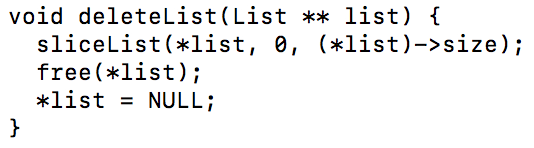
removeList удаляет первую попавшуюся цель с начала списка. Заводим tmp, бежим до совпадения, если нет такого элемента, ничего не делаем, иначе обмениваемся с соседями ссылками, вырезаем из списка target, удаляем его, удаляем tmp. Здесь я для разнообразия использовал slice вместо pop и shift, потому что для них необходимо создавать переменные куда будет возвращен результат. Была идея просто найти индекс первого включения и передать его в slice, но при больших размерах листа, это увеличит чисто итераций пробега.



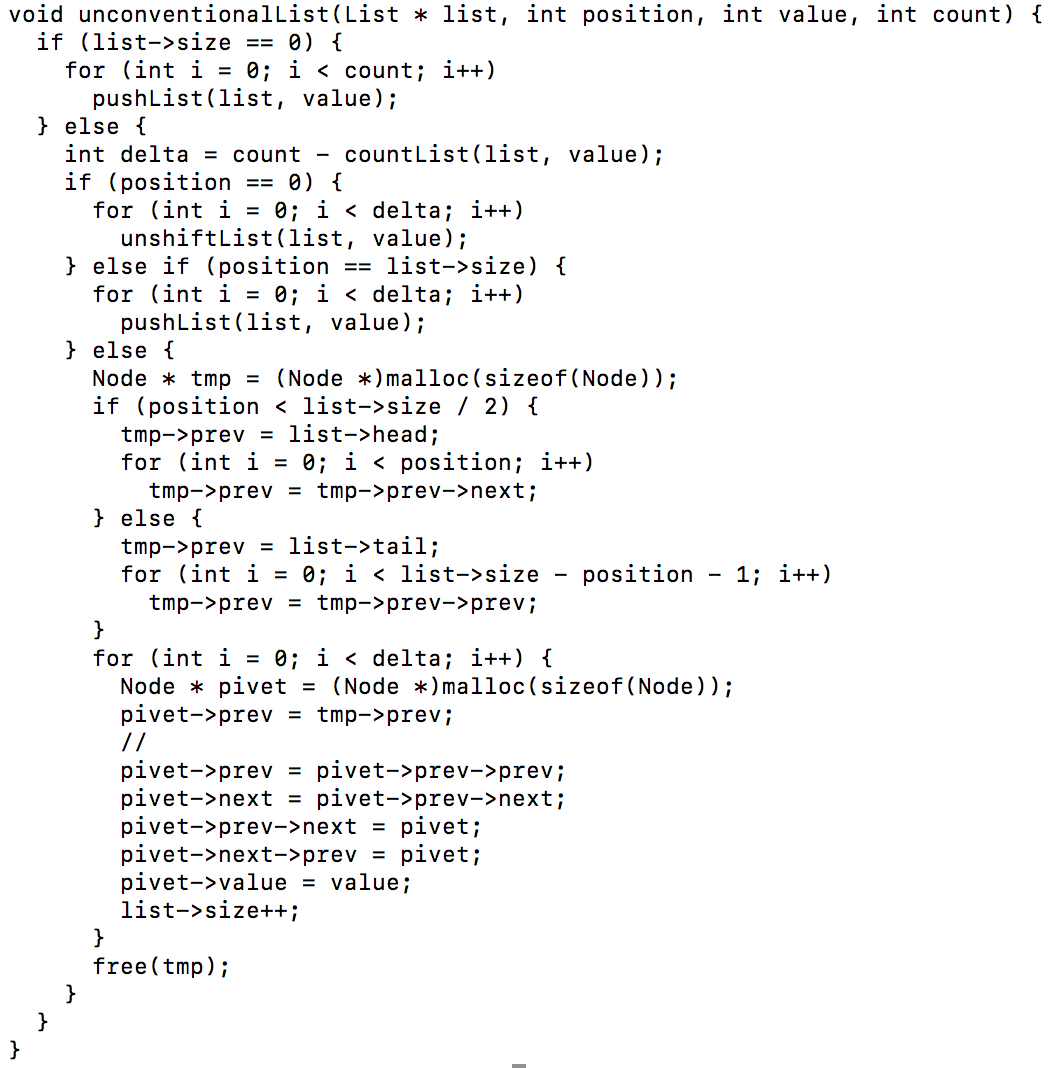
countList считает количество включений данного значения в списке. Также создает tmp, который бежит, сравнивая значения с target, при соответствии инкрементирует счетчик.



clearList очищает весь списко от значений, удаляя их соответственно. slice тут подходит идеально.



deleteList полностью удаляет и освобождает память выделенную под список, удаляя каждый node, обнуляя в ноль ссылку list.



unconventionalList соответствующей функции соответствующее название, та самая дополнительная функция для списка.  
Если пустой список, пушим k значений, если нет, начинаем считать дельту, сколько надо еще добавить, чтобы добить до количества k. Если позиция вставки соответствует краям листа, push или unshift его в конец или начало соответственно. Если не так повезло, заводим tmp, шагаем слева или справа, смотря откуда ближе к нужному индексу и в цикле выделяем node и вставляем его количество раз delta. Удаляем tmp.

На этом заканчиваются функции листа.

**main.c**

#include <stdio.h>

#include <stdlib.h>

#include "list.h"

char q;

int pass = 1;

int req, start, count;

int main() {

List \* list = createList();

//

printf("1 for push\n2 for pop\n3 for shift\n4 for unshift\n5 for incert\n6 for index\n7 for delete target\ns for slice\np for print\nu for unconventional\nc for clear\n");

while(pass) {

scanf("%c", &q);

switch (q) {

case ('1'):

printf("enter: ");

scanf("%d", &req);

pushList(list, req);

break;

case ('2'):

printf("res: %d\n", popList(list));

break;

case ('3'):

printf("res: %d\n", shiftList(list));

break;

case ('4'):

printf("enter: ");

scanf("%d", &req);

unshiftList(list, req);

break;

case ('5'):

printf("position: ");

scanf("%d", &start);

printf("value: ");

scanf("%d", &count);

insertList(list, start, count);

break;

case ('6'):

printf("target: ");

scanf("%d", &count);

printf("index: %d\n", indexList(list, count));

break;

case ('7'):

printf("target: ");

scanf("%d", &count);

removeList(list, count);

break;

case ('r'):

reverseList(list);

break;

case ('s'):

printf("start: ");

scanf("%d", &start);

printf("count: ");

scanf("%d", &count);

sliceList(list, start, count);

break;

case ('p'):

printf("list: ");

printList(list);

printf("size: %d\n", sizeList(list));

break;

case ('u'):

printf("position: ");

scanf("%d", &start);

printf("value: ");

scanf("%d", &req);

printf("count: ");

scanf("%d", &count);

unconventionalList(list, start, req, count);

break;

case ('c'):

clearList(list);

break;

case ('q'):

deleteList(&list);

pass = 0;

break;

}

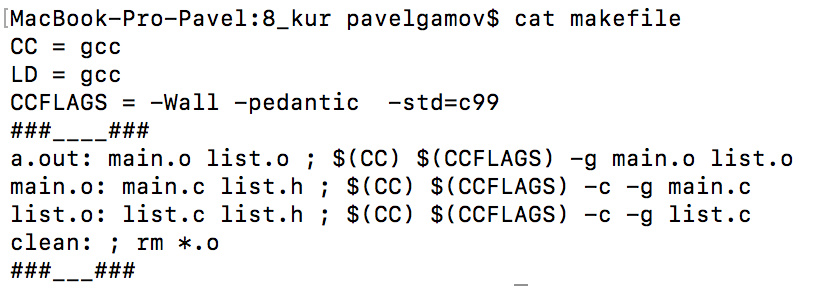
}

return 0;

}

Самое обычное меню с командами.

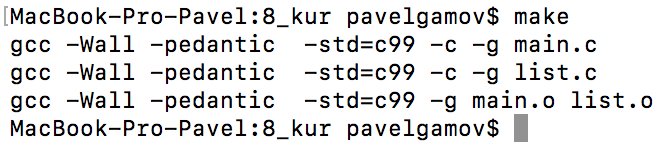
**makefile**

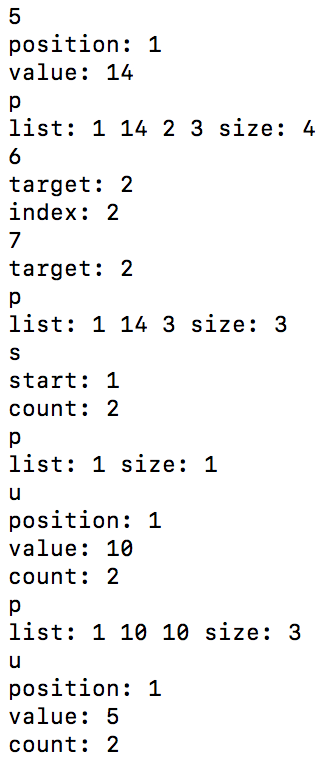
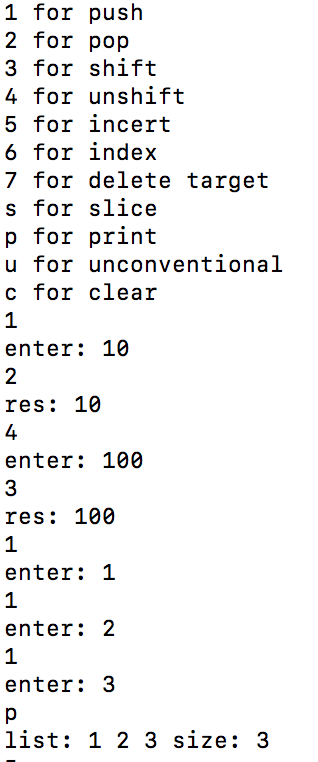


Утилита, которая полюбилась мне с 26 лабораторной работы.

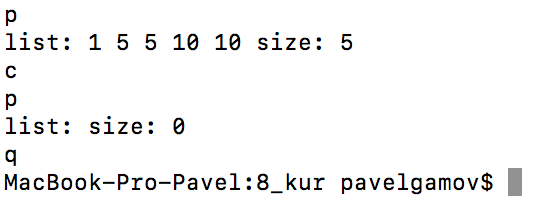
Ключ -g нужен для дальнейшей отладки или пропускания через valgrind.

**Пример работы**



\/\/\/\/\/\/\/\/

>>>>>>



**Заключение**

Использования динамической памяти позволяет распоряжаться ей как вздумается пользователю, это дает возможность реализовывать абстрактные типы данных по типу списка, нужных для различных целей в программировании. Система указателей позволяет объектам ссылаться на себе подобные, позволяя распоряжаться памятью более экономно, беря ее маленькими частями.

Особенно данная реализация полезна при невозможности выделить одну полоску памяти под массив соответствующих размеров.

Вместе с этим появляются проблемы утечек памяти, которые надо контролировать.

**Список использованной литературы**

Указатель в языке Си – <https://prog-cpp.ru/c-pointers/>

Абстрактные типы данных – <https://prog-cpp.ru/cpp-atd/>

Абстрактные типы данных – <https://ru.wikipedia.org/wiki/Абстрактный_тип_данных>

список на Си – https://codedream.me/2016/10/12/односвязный-список/