Федеральное государственное автономное образовательное учреждение высшего образования
РОССИЙСКИЙ УНИВЕРСИТЕТ ДРУЖБЫ НАРОДОВ
Инженерная академия
Кафедра механики и процессов управления

КУРСОВАЯ РАБОТА

«РЕАЛИЗАЦИЯ ОДНОСВЯЗНОГО СПИСКА НА ЯЗЫКЕ СИ»

направление подготовки «Управление в технических системах». Квалификация «бакалавр».

Разработчик **Шуркин И. А.**
Государство РФ
Студент группы ИУСбд-02-23
Студенческий билет № 1132233614

Научный руководитель

Оценка. ECTS_____ Балл _____

Оригинальность в "Антиплагиат" _____ %
проставляется научным руководителем

Москва
2024

СОДЕРЖАНИЕ

ВВЕДЕНИЕ 3
ГЛАВА I. ТЕОРИЯ 4
1. Основные понятия 4
2. Односвязный список 4
3. Сравнение с массивами 5
ГЛАВА II. ПРАКТИКА 6
1. Алгоритмы основных операций 6
2. Методы тестирования 6
ЗАКЛЮЧЕНИЕ 8
СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ 9

ВВЕДЕНИЕ

Современные информационные технологии требуют эффективной организации и обработки данных. Именно от выбора правильной и качественной структуры данных зависит производительность и надежность программного обеспечения. Среди множества структур данных, связанный список занимает особое место благодаря своей гибкости и простоте реализации, что делает его незаменимым инструментом в различных областях программирования. В данной курсовой работе рассматривается реализация связанного списка на языке программирования Си.

Актуальность: связанные списки широко применяются в решении задач, где требуется динамическое изменение размера данных. По сравнению с массивами, связанные списки имеют такое преимущество, как динамическое распределение памяти, что дает возможность эффективно работать с данными переменного размера.

Цель: разработка и реализация библиотеки функций для работы с односвязным списком на языке Си, которая включает в себя такие операции как: создание, добавление, удаление, поиск и вывод элементов.

Основные задачи:

- Разработка структуры данных узла списка
- Реализация алгоритмов основных операций
- Тестирование реализованных функций
- Оценка эффективности

Результаты работы могут быть использованы в качестве основы для дальнейшей разработки более сложных структур данных, а также при решении практических задач, требующих использования динамических структур данных.

ГЛАВА I. ТЕОРИЯ

1. Основные понятия

Структура данных – это программная единица, описывающая способ организации ячеек памяти, хранящих данные одного и того же типа.
Временная сложность – зависимость выполнения операции от количества элементов в структуре данных (обычно выражается в нотации "Большое O"). Например, O(1) означает постоянное время, независимое от размера данных, O(n) — линейное время, зависящее от размера данных, O(log n) — логарифмическое время и т.д.
Классификация структур данных:
По сложности:
- Простые (базовые, примитивные);
- Интегрированные (композитные, сложные).

По связи между элементами:
- Связные (связные списки);
- Несвязные (векторы, массивы, строки, очереди).

Линейные структуры
по расположению элементов в памяти:
- Последовательные (векторы, строки, массивы, стеки, очереди);
- Произвольные (односвязные, двусвязные списки).

Нелинейные структуры
- Многосвязные списки, деревья, графы.

2. Односвязный список

Односвязный список — это линейная динамическая структура данных, элементы которой (узлы) не хранятся в непрерывной области памяти, а связаны между собой с помощью указателей. Каждый узел списка содержит два основных компонента:
• Данные: Информация, хранящаяся в узле. Тип данных может быть любым (integer, float, char, struct и т.д.).
• Указатель (ссылка): Адрес памяти следующего узла в списке. Последний узел списка имеет указатель, равный NULL (или 0), сигнализирующий об окончании списка.

Преимущества:
Динамическое распределение памяти: память выделяется по мере необходимости, что позволяет эффективно работать с данными переменного размера.
Быстрая вставка и удаление элементов: вставка или удаление узла в произвольном месте списка требует изменения указателей только соседних узлов.
Недостатки:
Доступ к элементам по индексу медленный: для доступа к i-ому элементу

необходимо пройти по списку от начала до i-го элемента.
Дополнительная память на указатели: для хранения указателей требуется дополнительная память.
Невозможность обратного прохода: Перемещение по списку возможно только в одном направлении.

3. Сравнение с массивами

Массивы и односвязные списки — две основные линейные структуры данных. Они отличаются по способу хранения данных и, следовательно, по эффективности выполнения различных операций.

Характеристика	**Массив**	**Односвязный список**
Размер	фиксированный	динамический
Использование памяти	непрерывный блок памяти	непрерывный блок памяти не требуется
Доступ к элементам	O(1)	O(n)
Вставка/удаление элемента	O(n)	O(1), при условии знания позиции

Выбор между массивом и односвязным списком зависит от конкретной задачи. Если необходим частый доступ к элементам по индексу, то предпочтительнее массив. Если требуется частое добавление и удаление элементов, то лучше использовать односвязный список.

ГЛАВА II. ПРАКТИКА

1. Алгоритмы основных операций

В этой части опишем алгоритмы основных операций над односвязным списком. Предполагается, что узел списка описывается структурой:
1. Создание списка (create_list()):
Алгоритм тривиален: возвращается указатель на NULL, представляющий пустой список.
2. Добавление элемента (add_element()):
Существуют три основных варианта добавления элемента: в начало, в конец и по индексу.

- Добавление в начало: создается новый узел, его next указывает на текущую голову списка, а новый узел становится новой головой.
- Добавление в конец: необходимо пройти по всему списку до последнего узла, а затем добавить новый узел после него.
- Добавление по индексу: требуется найти узел с указанным индексом и вставить новый узел перед ним. Обработка случаев, когда индекс вне диапазона, необходима.

3. Удаление элемента (delete_element()):

- Удаление по значению: необходимо пройти по списку и найти узел с заданным значением. Затем нужно изменить указатель предыдущего узла, чтобы он указывал на узел, следующий за удаляемым.
- Удаление по индексу: находим узел с указанным индексом и удаляем его, аналогично удалению по значению. Обработка ошибок необходима.

4. Поиск элемента (search_element()):
Проходим по списку и сравниваем данные каждого узла с искомым значением. Возвращается указатель на найденный узел или NULL, если узел не найден.
5. Вывод списка (print_list()):
Проходим по списку и выводим данные каждого узла.
6. Освобождение памяти (free_list()):
Необходимо освободить память, выделенную для каждого узла списка, чтобы избежать утечек памяти. Проходим по списку, освобождая память для каждого узла.

2. Методы тестирования

1. Модульное тестирование:
Этот метод предполагает тестирование каждой функции библиотеки независимо от остальных. Цель — убедиться, что каждая функция работает правильно в изоляции. Для этого для каждой функции будут разработаны тестовые случаи, покрывающие различные сценарии её использования:

• Тестирование функции create_list(): Проверка успешного создания пустого списка.
• Тестирование функции add_element(): Добавление элемента в начало, в конец и в середину списка; проверка корректности связей между узлами после добавления; обработка случая добавления в пустой список.
• Тестирование функции delete_element(): Удаление первого, последнего и произвольного элемента; удаление элемента из пустого списка; удаление единственного элемента; проверка корректности связей после удаления.
• Тестирование функции search_element(): Поиск существующего элемента; поиск несуществующего элемента; поиск в пустом списке; поиск первого и последнего элемента.
• Тестирование функции print_list(): Вывод содержимого списка различной длины; вывод пустого списка; проверка корректности вывода элементов в нужном порядке.
• Тестирование функции free_list(): Проверка освобождения памяти, занимаемой всеми узлами списка; предотвращение утечек памяти.
Для каждого тестового случая будут определены ожидаемые результаты, и результаты выполнения функции будут сравниваться с ожидаемыми. В случае несовпадения будет регистрироваться ошибка.

2. Интеграционное тестирование:
После успешного модульного тестирования выполняется интеграционное тестирование, которое проверяет взаимодействие между различными функциями библиотеки. Цель — убедиться, что функции корректно работают вместе.
Тестирование будет включать в себя:
• Последовательность операций: Проверка корректной работы последовательности операций над списком (например, добавление, затем поиск, затем удаление элементов).
• Граничные случаи: Проверка работы функций на граничных значениях (например, добавление/удаление элемента в пустой или почти пустой список).
• Обработка ошибок: Проверка корректной обработки ошибок (например, попытка удалить несуществующий элемент, доступ к элементу за пределами списка).

3. Системное тестирование (опционально):
Если разработанная библиотека является частью более крупной системы, то выполняется системное тестирование, которое проверяет работу библиотеки в контексте всей системы. Это тестирование выходит за рамки данной курсовой работы, но может быть упомянуто в качестве перспективы дальнейшего развития.

ЗАКЛЮЧЕНИЕ

В данной курсовой работе была успешно реализована библиотека функций для работы с односвязным списком на языке программирования Си. Были разработаны и протестированы функции создания списка, добавления, удаления и поиска элементов, а также функция вывода содержимого списка и освобождения занимаемой им памяти. В ходе работы были изучены основные принципы работы с динамическими структурами данных, особенности реализации односвязного списка и методы тестирования программного обеспечения.

Разработанная библиотека прошла модульное и интеграционное тестирование, подтвердившее корректность работы всех функций в различных сценариях, включая граничные условия. Результаты тестирования показали эффективность реализованных алгоритмов, подтвердив теоретические оценки временной сложности операций.

В процессе работы над курсовой были получены практические навыки программирования на языке Си, углублено понимание принципов работы динамических структур данных и методов их тестирования. Разработанная библиотека может быть использована в качестве основы для создания более сложных структур данных или при решении различных практических задач, требующих использования динамических списков.

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Кнут, Д. Искусство программирования. Том 1. Основные алгоритмы. – М.: Вильямс, 2006.
2. Вирт, Н. Алгоритмы и структуры данных. – СПб.: Невский Диалект, 2006.
3. Седжвик, Р. Алгоритмы на языке С. Части 1-4. – 3-е изд. – СПб.: Невский Диалект, 2002.
4. Овсянников А. В., Пикман Ю. А. Алгоритмы и структуры данных – Минск: БГУ, 2015.

ПРИЛОЖЕНИЕ

Программный код основных функций:

```
#include <stdio.h>

typedef struct node{
    int data;
    struct node *next;
} Node;

Node* create_list(){
    return NULL;
}

void add_element(Node** head, int index, int data) {
if (index < 0) {    // Обработка ошибок: индекс вне диапазона
    return;
}

Node* new_node = (Node*)malloc(sizeof(Node));
new_node->data = data;

if (index == 0) {
new_node->data = data;
new_node->next = *head;
*head = new_node;
return;
}

Node* current = *head;
int i = 0;
while (current != NULL && i < index) {
    current = current->next;
    i++;
}

if (current == NULL) {      // Обработка ошибки: индекс вне диапазона
    free(new_node);
    return;
}

new_node->next = current->next;
current->next = new_node;
}

void delete_element_value(Node** head, int data) {
Node* current = *head;
```

```
Node* prev = NULL;

while (current != NULL && current->data != data) {
    prev = current;
    current = current->next;
}

if (current == NULL) return; // Элемент не найден

if (prev == NULL) {
    *head = current->next; // Удаление первого элемента
}
else {
    prev->next = current->next;
}

free(current);
}

Node* search_element(Node* head, int data) {
Node* current = head;
while (current != NULL) {
    if (current->data == data) {
        return current;
    }
    current = current->next;
}
return NULL;
}

void print_list(Node* head) {
Node* current = head;
while (current != NULL) {
    printf("%d ", current->data);
    current = current->next;
}
printf("\n");
}

void free_list(Node** head) {
Node* current = *head;
Node* next;
while (current != NULL) {
    next = current->next;
    free(current);
    current = next;
}
*head = NULL;
}

int main()
```

```
{
    Node *head = create_list();
    add_element(&head, 0, 10);
    add_element(&head, 0, 11);
    add_element(&head, 0, 12);
    delete_element_value(&head, 11);
    print_list(head);
    free_list(0);
}
```