Отчет по лабораторной работе № 23 по курсу
Алгоритмы и структуры данных

Студент группы М8О-103Б-22 Ахметшин Булат Рамилевич, № по списку 2

Контакты www, e-mail, icq, skype ahmbulat04@yandex.ru

Работа выполнена: 21.04.2023 г.

Преподаватель: доцент каф. 806 Никулин С.П.

Входной контроль знаний с оценкой _____

Отчет сдан « » _____ 202 __ г., итоговая оценка ___

Подпись преподавателя _____

1. **Тема:** Динамические структуры данных, обработка деревьев. _____

2. **Цель работы:** Научиться реализовывать динамические структуры данных, такие как деревья, на языке программирования Си и работать. _____

3. **Задание** (*вариант № 2*): Составить программу на языке Си для построения и обработки упорядоченного двоичного дерева, содержащего узлы типа int. _____

4. **Оборудование** (*лабораторное*):
ЭВМ _____, процессор _____, имя узла сети _____ с ОП _____ Мб,
НМД _____ Мб. Терминал _____ адрес _____. Принтер _____
Другие устройства _____

Оборудование ПЭВМ студента, если использовалось:
Процессор Intel(R) Core(TM) i7-10510U с ОП 8 ГБ НМД SSD 512 ГБ . Монитор Встроенный 1920x1080
Другие устройства _____

5. **Программное обеспечение** (*лабораторное*):
Операционная система семейства _____, наименование _____ версия _____
интерпретатор команд _____ версия _____
Система программирования _____ версия _____
Редактор текстов _____ версия _____
Утилиты операционной системы _____
Прикладные системы и программы _____
Местонахождение и имена файлов программ и данных _____

Программное обеспечение ЭВМ студента, если использовалось:
Операционная система семейства UNIX, наименование Ubuntu версия 22.04
интерпретатор команд GNU bash версия 5.1.16
Система программирования Visual Studio Code версия 1.77.3
Редактор текстов Sublime Text 3 версия 3211

Утилиты операционной системы Стандартные утилиты OS Linux
Прикладные системы и программы Редактор текста nano.
Местонахождение и имена файлов программ и данных на домашнем компьютере /home/bulat/Studying/prprm/l/l23

6. **Идея, метод, алгоритм** решение задачи (в формах: словесной, псевдокода, графической [блок-схема, диаграмма, рисунок, таблица] или формальные спецификации с пред- и постусловиями)

Основные структуры программы будут описаны следующим образом:

imap - map, реаилзованный на основе двустороннего ассоциативного списка из структур imap_item

```
typedef struct imap {
    imap_item* begin;
    imap_item* back;
    uint64_t size;
} imap;
```

imap_item - структуры, соддержащие в себе ключ, значение, ссылку на следующий и предыдущий элемент

```
typedef struct imap_item {
    int64_t id;
    int64_t value;
    struct imap_item* prev;
    struct imap_item* next;
} imap_item;
```

node - структура, реализующая узел бинарного дерева

```
typedef struct node {
    // Identification number of each node
    uint64_t id;
    // Data of node
    int64_t data;
    // Pointer to left node of b_three
    struct node* left;
    // Pointer to right node of b_tree
    struct node* right;
    // Pointer to node's ancestor
    struct node* p;
} node;
```

b_tree - реализация бинарного дерева

```
typedef struct b_tree{
    // Count of nodes
    int64_t n;
    // Id of last added node
    int64_t last_id;
    // Depth of each node
    imap d;
    // Degree of each node
    imap c;
    // Pointer to root node of b_tree
    node* root;

} b_tree;
```

7. **Сценарий выполнения работы** (план работы, первоначальный текст программы в черновике [можно на отдельном листе] и тесты либо соображения по тестированию)

   (a) Реализовать последовательную структуру данных map, с ключом-идентификатором и целочисленным значением

   (b) Реализовать структуру данных бинарное дерево, вершинами которого будут являться структуры типа node - узлы с идентификаторами, значениями и указателями на предка, младшего и старшего братьев.

   (c) Релизовать структуру данных строка, написать алгоритмы парсинга поступающих комманд

   (d) Написать итоговый код программы, протестировать

Пункты 1-7 отчета составляются строго до начала лабораторной работы.
Допущен к выполнению работы. **Подпись преподавателя** _____

8. **Распечатка протокола** (подклеить листинг окончательного варианта программы с тестовыми примерами, подписанный преподавателем)

```
bulat@bulat-Swift-SF314-58:~/Studying/prprm/l/l23$ script logs/proto_1
Script started, output log file is 'logs/proto_1'.
bulat@bulat-Swift-SF314-58:~/Studying/prprm/l/l23$ ls
imap.c  imap.h  logs  main  main.c  README.md  string.c  string.h  tree.c  tree.h
bulat@bulat-Swift-SF314-58:~/Studying/prprm/l/l23$ cat imap.h
#ifndef M_IMAP_H

#define M_IMAP_H

#include <inttypes.h>
#include <stdlib.h>
#include <stdio.h>

typedef struct imap_item {
    int64_t id;
    int64_t value;
    struct imap_item* prev;
    struct imap_item* next;
} imap_item;

typedef struct imap {
    imap_item* begin;
    imap_item* back;
    uint64_t size;
} imap;

imap_item* new_imap_item(int64_t id, int64_t value, \
                         imap_item* const prev, imap_item* const next);

imap_item* copy_imap_item(imap_item* const i);

imap create_empty_imap();

int64_t imap_at(imap* const m, int64_t id);

void imap_add(imap* m, int64_t id, int64_t value);

void imap_remove_by_id(imap* m, int64_t id);

void imap_remove_by_value(imap* m, int64_t value);

int8_t is_empty_imap(imap* const m);

int8_t is_valid_edges(imap* const m);

int8_t is_valid(imap* const m);

void make_valid(imap* m);
```

```
void clear_imap(imap* m);

void imap_copy(imap* m, imap* const m_);

#define mat(m, id) imap_at((m), (id))

#define log_exception(M, ...)                                                \
{                                                                            \
    fprintf(stderr, "(%s:%d) " M "\n", __FILE__, __LINE__, ##__VA_ARGS__); \
}

#define exception_exit(M, ...)           \
{                                        \
    log_exception(M, ##__VA_ARGS__);     \
    exit(-1);                            \
}


#endifbulat@bulat-Swift-SF314-58:~/Studying/prprm/l/l23$ cat imap.c
#include "imap.h"

imap_item* new_imap_item(int64_t id, int64_t value, imap_item* const prev, imap_item* const next) {
    imap_item* i = (imap_item*)malloc(sizeof(imap_item));
    i->id = id; i->value = value;
    i->prev = prev;
    i->next = next;
}

imap_item* copy_imap_item(imap_item* const i) {
    return new_imap_item(i->id, i->value, i->prev, i->next);
}

imap create_empty_imap() {
    imap m;
    m.begin = NULL;
    m.back = NULL;
    m.size = 0;
    return m;
}

int64_t imap_at(imap* const m, int64_t id) {
    imap_item* iterator = m->begin;
    if (iterator == NULL) {
        exception_exit("imap at address \'%p\' begin is NULL\n", m);
    }
    while (!(iterator == NULL) && iterator->id != id) {
        iterator = iterator->next;
    }
    if (!(iterator == NULL) && iterator->id == id) {
        return iterator->value;
    }
    exception_exit("No element with id \'%ld\' in imap at address \'%p\'\n", id, m);
}

void imap_add(imap* m, int64_t id, int64_t value) {
    imap_item* i = new_imap_item(id, value, NULL, NULL);
    if (is_empty_imap(m)) {
        m->begin = m->back = i;
        m->size = 1;
        return;
    }
    if (m->begin == NULL) {
        exception_exit("Bad imap: imap at address \'%p\' begin is NULL\n", m);
    }
    if (m->back == NULL) {
        exception_exit("Bad imap: imap at address \'%p\' back is NULL\n", m);
    }
    imap_item* iterator = m->begin;
    while(!(iterator->next == NULL) && (i->id > iterator->id)) {
        iterator = iterator->next;
    }
    if (iterator->next == NULL) {
        i->prev = iterator;
        iterator->next = i;
        m->back = i;
    }
    else {
        iterator = iterator->next;
        if (iterator == m->begin) {
            i->next = iterator;
            iterator->prev = i;
            m->begin = i;
        } else {
            i->next = iterator;
            i->prev = iterator->prev;
```

```
            iterator->prev->next = i;
            iterator->prev = i;
        }
    }
    ++m->size;
}

void imap_remove_by_id(imap* m, int64_t id) {
    imap_item* iterator = m->begin;
    while (!(iterator == NULL) && iterator->id != id) {
        iterator = iterator->next;
    }
    if (!(iterator == NULL) && iterator->id == id) {
        if (iterator->prev == NULL) {
            m->begin = iterator->next;
        } else {
            iterator->prev->next = iterator->next;
        }
        if (iterator->next == NULL) {
            m->back = iterator->prev;
        } else {
            iterator->next->prev = iterator->prev;
        }
        free(iterator);
    }
}

void imap_remove_by_value(imap* m, int64_t value) {
    imap_item* iterator = m->begin;
    while (!(iterator == NULL) && iterator->value != value) {
        iterator = iterator->next;
    }
    if (!(iterator == NULL) && iterator->value == value) {
        if (iterator->prev == NULL) {
            m->begin = iterator->next;
        } else {
            iterator->prev->next = iterator->next;
        }
        if (iterator->next == NULL) {
            m->back = iterator->prev;
        } else {
            iterator->next->prev = iterator->prev;
        }
        free(iterator);
    }
}

void imap_copy(imap* m, imap* const m_) {
    if (!is_valid_edges(m)) {
        exception_exit("imap \'%p\' is not valid to copy\n", m);
    }
    clear_imap(m);
    imap_item* iterator_ = m_->begin;
    m->begin = copy_imap_item(iterator_);
    iterator_ = iterator_->next;
    imap_item* iterator  = m->begin;
    while (!(iterator_ == NULL)) {
        iterator->next = copy_imap_item(iterator_);
        iterator = iterator->next;
    }
    m->back = iterator;
}

int8_t is_valid_edges(imap* const m) {
    return !((m->begin == NULL || m->back == NULL) && !(m->begin == m->back));
}

int8_t is_valid(imap* const m) {
    if (!is_valid_edges(m)) {
        return 0;
    }
    int8_t res = 1;
    imap_item* iterator = m->begin;
    while (!(iterator == NULL) && iterator != m->back) {
        iterator = iterator->next;
    }
    return iterator == m->back;
}

void make_valid(imap* m) {
    imap_item* l = m->begin,* r = m->back;
    uint64_t count = 0;
    while (!(l == NULL) && !(l->next == NULL)) {
        l = l->next;
        ++count;
```

```
    }
    while (!(r == NULL) && !(r->prev == NULL)) {
        r = r->prev;
        ++count;
    }
    if (l == NULL || r == NULL) {
        if (l == NULL) {
            m->begin = r;
        }
        if (r == NULL) {
            m->back = l;
        }
    } else {
        l->next = r;
        r->prev = l;
    }
    m->size = count;
}

void clear_imap(imap* m) {
    imap_item* iterator = m->begin;
    if (iterator == NULL) {
        iterator = m->back;
        if (iterator == NULL) {
            m->size = 0;
            return;
        } else {
            while (!(iterator->prev == NULL)) {
                iterator = iterator->prev;
                free(iterator->next);
            }
        }
    } else {
        while (!(iterator->next == NULL)) {
            iterator = iterator->next;
            free(iterator->prev);
        }
    }
    free(iterator);
    m->begin = m->back = NULL;
    m->size = 0;
}

int8_t is_empty_imap(imap* const m) {
    return m->begin == m->back && m->back == NULL;
}bulat@bulat-Swift-SF314-58:~/Studying/prprm/l/l23$ cat string.h
#ifndef STRING_H

#define STRING_H

#define ADD_MEMORY_COEFFICIENT 2
#define INIT_VALUE ' '

#include <inttypes.h>
#include <stdio.h>
#include <stdlib.h>

// Структура Строка
struct string {
char* s;
uint64_t memory_size;
uint64_t last_char;
};

typedef struct string string;

// Процедура конструирования строки из одного символа INIT_VALUE
void construct_empty(string* s);

string not_allocated_string();

// Процедура конструирования строки из n символов
void construct_from_n(string* s, uint64_t n);

// Процедура конструирования строки из другой строки
void construct_from_s(string* s, string s_);

// Процедура конструирования строки из массива символов
void construct_from_char_pointer(string* s, const char* p);

// Процедура деструктирования строки
void destruct(string* s);

// Аннулировать строку
void annul(string* s);
```

```
// Процедура присвоения памяти
void appropriate_memory(string* s, string* s_);

// Процедура присвоения строки
void appropriate_string(string* s, string s_);

// Процедура копирования строки (без учёта памяти)

// Пример: <'abc'; '1234'> -> <'123'; '1234'>
// Пример: <'abcd'; '123'> -> <'123 '; '123'>
void copy_string(string* s, string s_);

// Процедура добавления памяти в строку
void add_memory(string* s, uint64_t n);

// Процедура добавления символа в строку
void add_char(string* s, char c);

// Конкатенация строк
void add_string(string* s, string s_);

int8_t equal_string(string* const s, string* const s_);

int8_t equal_charp(string* const s, const char* p);

// Считать строку
int8_t read_line(string* s);

// Считать все строки
void read(string* s);

// Вывести строку
void print(string s);

// Получить i-тый символ строки
char* at(string* s, uint64_t i);

// Макросы

// Максимум
#define max(a, b) (a > b ? a : b)
// Минимум
#define min(a, b) (a < b ? a : b)
// Логирование
#define log_info(M, ...) fprintf(stderr, "(%s:%d) " M "\n",\
        __FILE__, __LINE__, ##__VA_ARGS__)

#endifbulat@bulat-Swift-SF314-58:~/Studying/prprm/l/l23$ cat string.c
#include "string.h"

// Процедура конструирования строки из одного символа ' '
void construct_empty(string* s) {
    construct_from_n(s, 1);
}

string not_allocated_string() {
string s;
annul(&s);
return s;
}

// Процедура конструирования строки из n символов
void construct_from_n(string* s, uint64_t n) {
    // Один последний символ зарезервирован под '\0'
s->s = (char*)malloc(sizeof(char) * n + 1);
if (!(s->s == NULL)) {
s->memory_size = n;
s->last_char = 0;
        // Строка инициализирована символом INIT_VALUE, объявленном в string.h
for (uint64_t i = 0; i < n; ++i)
s->s[i] = INIT_VALUE;
        s->s[n] = '\0';
}
    else {
        log_info("REFUZE_MEMORY_ALLOCATION\n");
        exit(-1);
    }
}

// Процедура конструирования строки из другой строки
void construct_from_s(string* s, string s_) {
construct_from_n(s, s_.memory_size);
copy_string(s, s_);
}
```

```
// Процедура конструирования строки из массива символов
void construct_from_char_pointer(string* s, const char* p) {
if (p == NULL) {
        log_info("NULL_POINTER_ACCES\n");
        exit(-1);
    }
construct_from_n(s, 1);
uint64_t i = 0;
while (p[i] != '\0') {
add_char(s, p[i]);
++i;
}
    add_char(s, '\0');
}

// Процедура деструктирования строки
void destruct(string* s) {
if (s->s) {
free(s->s);
s->last_char = 0;
s->memory_size = 0;
}
    else {
    log_info("DEALLOCATE_NULL_POINTER");
        exit(-1);
    }
}

// Аннулировать строку
void annul(string* s) {
s->s = NULL;
s->last_char = 0;
s->memory_size = 0;
}

// Процедура присвоения памяти
void appropriate_memory(string* s, string* s_) {
destruct(s);
s->s = s_->s;
s->memory_size = s_->memory_size;
s->last_char = s_->last_char;
annul(s_);
}

// Процедура копирования строки (без учёта памяти)

// Пример:
// <'abc', '1234'> -> <'123', '1234'>
// <'abcd', '123'> -> <'123 ', '123'>
// <'abc',  '123'> -> <'123',  '123'>
void copy_string(string* s, string s_) {
if (s->s && s_.s) {
uint64_t i = 0;
while (i < s->memory_size && i < s_.last_char) {
s->s[i] = s_.s[i];
++i;
}
while (i < s->memory_size) {
s->s[i] = ' ';
++i;
}
        s->last_char = min(s->memory_size, s_.last_char);
}
else {
        log_info("NULL_POINTER_ACCES");
        exit(-1);
    }
}

// Процедура присвоения строки
void appropriate_string(string* s, string s_) {
construct_from_s(s, s_);
copy_string(s, s_);
}


// Процедура добавления памяти в строку
void add_memory(string* s, uint64_t n) {
string s_;
construct_from_n(&s_, s->memory_size * n);
copy_string(&s_, *s);
appropriate_memory(s, &s_);
}
```

```
// Добавить символ
void add_char(string* s, char c) {
if (s->s) {
if (s->last_char >= s->memory_size) {
add_memory(s, ADD_MEMORY_COEFFICIENT);
}
s->s[s->last_char] = c;
        if (c != '\0')
            s->last_char++;
s->s[s->last_char] = '\0';
}
else {
        log_info("NULL_POINTER_ACCES");
        exit(-1);
    }
}

// Конкатенация строк
void add_string(string* s, string s_) {
    string S;
    construct_from_n(&S, s->last_char + s_.last_char);
    while (S.last_char < s->last_char) {
        S.s[S.last_char] = s->s[S.last_char];
        S.last_char++;
    }
    while (S.last_char - s->last_char < s_.last_char) {
        S.s[S.last_char] = s_.s[S.last_char - s->last_char];
        S.last_char++;
    }
    add_char(&S, '\0');
    appropriate_memory(s, &S);
}

int8_t equal_string(string* const s, string* const s_) {
if (s->last_char != s_->last_char) {
return 0;
}
for (uint64_t i = 0; i < s->last_char; ++i) {
if (*at(s, i) != *at(s_, i))
return 0;
}
return 1;
}

int8_t equal_charp(string* const s, const char* p) {
string s_;
construct_from_char_pointer(&s_, p);
return equal_string(s, &s_);
}

// Считать строку
int8_t read_line(string* s) {
char c;
while (1) {
//scanf("%c", &c);
c = getchar();
        if (c == EOF || c == '~') {
            add_char(s, '\0');
            return 0; // Конец потока
        }
        if (c == '\n') {
add_char(s, c);
            return 1; // Конец строки
        }
add_char(s, c);
}
}

// Считать все строки
void read(string* s) {
    while (read_line(s)) {}
}

// Вывести строку
void print(string s) {
printf("%s", s.s);
}

// Получить i-тый символ строки
char* at(string* s, uint64_t i) {
if (i < s->memory_size) {
return &s->s[i];
    }
    else {
    log_info("OCCUPIED_MEMORY_ACCES");
```

```
        exit(-1);
    }
}bulat@bulat-Swift-SF314-58:~/Studying/prprm/l/l23$ cat tree.h
#ifndef M_TREE_H

#define M_TREE_H

#include <inttypes.h>
#include <stdio.h>
#include <stdlib.h>
#include "imap.h"

typedef enum {
    N_LEFT,
    N_RIGHT,
    N_ROOT
} SIDE;

// Inary tree node structure
typedef struct node {
    // Identification number of each node
    uint64_t id;
    // Data of node
    int64_t data;
    // Pointer to left node of b_three
    struct node* left;
    // Pointer to right node of b_tree
    struct node* right;
    // Pointer to node's ancestor
    struct node* p;
} node;

// Binary tree
typedef struct b_tree{
    // Count of nodes
    int64_t n;
    // Id of last added node
    int64_t last_id;
    // Depth of each node
    imap d;
    // Degree of each node
    imap c;
    // Pointer to root node of b_tree
    node* root;

} b_tree;

// Creates empty b_tree
// {
//      n = 0
//      last_id = -1
//      d = create_empty_vector()
//      c = create_empty_vector()
//      root = NULL
// }
b_tree create_empty_b_tree();

void copy_b_tree(b_tree* t, b_tree* const t_);

void copy_b_tree_n_update(b_tree* t, b_tree* const t_);

void __copy_b_subtree(node** root, node* const to_copy, node* const acc);

void init_b_tree(b_tree* t, int64_t data);

// Checks if root pointer is NULL
int8_t is_empty_b_tree(b_tree* const t);

// Adds node py pointer to its ancestor by ancestors id
void add_node_by_node(b_tree* t, node* p, int64_t data, SIDE s);

// Adds node py pointer to its ancestor by ancestors id
void add_node_by_id(b_tree* t, int64_t id, int64_t data, SIDE s);

//      Deletes node by id
void delete_node_by_id(b_tree* t, int64_t id);

//      Deletes node by pointer
void delete_node_by_node(b_tree* t, node* v);

// Returns pointer to node of b_tree by id
node* node_by_id(node* const root, int64_t id);

// Returns id node data
int64_t data_by_id(b_tree* const t, int64_t id);
```

```
// Checks if given node is in b_tree
int8_t node_is_in_b_tree(node* const root, node* const v);

// Removes all nodes from b_tree and sets it empty
void clear_tree(b_tree* t);

// Node
// {
//     id = id;
//     data = data;
//     left = NULL;
//     right = NULL;
// }
node* new_node(int64_t id, int64_t data, node* const p);

void copy_node(node* v, node* const v_);

void update_depth(b_tree* t);

void __update_depth(imap* d, node* const root, int64_t depth);

void update_degree(b_tree* t);

void __update_degree(imap* c, node* const root);

void print_b_tree(node* const t, uint8_t tab, SIDE side);

int8_t is_B_tree(b_tree* const t);

#define log_exception(M, ...)                                           \
{                                                                       \
    fprintf(stderr, "(%s:%d) " M "\n", __FILE__, __LINE__, ##__VA_ARGS__); \
}

#define exception_exit(M, ...)          \
{                                       \
    log_exception(M, ##__VA_ARGS__);    \
    exit(-1);                           \
}

#endifbulat@bulat-Swift-SF314-58:~/Studying/prprm/l/l23$ cat tree.c
#include "tree.h"

b_tree create_empty_b_tree() {
    b_tree t;
    t.n = 0;
    t.last_id = -1;
    t.d = create_empty_imap();
    t.c = create_empty_imap();
    t.root = NULL;
    return t;
}

int8_t is_empty_b_tree(b_tree* const t) {
    return t->root == NULL;
}

void copy_b_tree(b_tree* t, b_tree* const t_) {
    clear_tree(t);
    __copy_b_subtree(&(t->root), t_->root, NULL);
    t->n = t_->n;
    t->last_id = t_->last_id;
}

void copy_b_tree_n_update(b_tree* t, b_tree* const t_) {
    copy_b_tree(t, t_);
    imap_copy(&(t->d), &(t_->d));
    imap_copy(&(t->c), &(t_->c));
}

void __copy_b_subtree(node** root, node* const to_copy, node* const acc) {
    if (to_copy == NULL)
        return;
    *root = new_node(to_copy->id, to_copy->data, acc);
    __copy_b_subtree(&(*root)->left, to_copy->left, *root);
    __copy_b_subtree(&(*root)->right, to_copy->right, *root);
}

void init_b_tree(b_tree* t, int64_t data) {
    if (!is_empty_b_tree(t)) {
        exception_exit("Initializing not empty b_tree");
    }
    t->root = new_node(0, data, NULL);
    t->last_id = 0;
```

```
    t->n = 1;
    t->d = create_empty_imap();
    imap_add(&(t->d), 0, 0);
    imap_copy(&(t->c), &(t->d));
}

void add_node_by_id (b_tree* t, int64_t id, int64_t data, SIDE s) {
    // Create new node
    if (is_empty_b_tree(t)) {
        init_b_tree(t, data);
    }
    node* v = node_by_id(t->root, id);
    if (v == NULL) {
        exception_exit("There is no node in tree %p with id %ld\n", t, id);
    }
    add_node_by_node(t, v, data, s);
}

void add_node_by_node(b_tree* t, node* p, int64_t data, SIDE s) {
    if (p == NULL) {
        exception_exit("Trying to access NULL node pointer\n");
    }
    node* v = p;
    // Add new node to ancestor with given id
    if (s == N_LEFT) {
        if (v->left == NULL) {
            v->left = new_node(t->last_id + 1, data, v);
        } else {
            exception_exit("%ld node already has left child - %ld node", v->id, v->left->id);
        }
    } else if (s == N_RIGHT) {
        if (v->right == NULL) {
            v->right = new_node(t->last_id + 1, data, v);
        }
        else {
            exception_exit("%ld node already has right child - %ld node", v->id, v->right->id);
        }
    } else {
        exception_exit("Root is already exist: add_node_by_node(%p, %p, %ld, N_ROOT), p id: %ld\n", t, p, data, p->id);
    }
    t->last_id++;
    t->n++;
}

node* node_by_id(node* const root, int64_t id) {
    node* const i = root;
    if (i == NULL || i->id == id) {
        return i;
    }
    node* l = i->left,* r = i->right;
    l = node_by_id(l, id);
    r = node_by_id(r, id);
    if (l != NULL) {
        return l;
    }
    if (r != NULL) {
        return r;
    }
    return NULL;
}

node* new_node(const int64_t id, int64_t data, node* const p) {
    node* v = (node*)malloc(sizeof(node));
    v->id= id;
    v->data = data;
    v->left = NULL;
    v->right = NULL;
    v->p = p;
    return v;
}

void copy_node(node* v, node* const v_) {
    v->data = v_->data;
    v->id = v_->id;
    v->left = NULL;
    v->right = NULL;
}

void delete_node_by_id(b_tree* t, const int64_t id) {
    node* v = node_by_id(t->root, id);
    delete_node_by_node(t, v);
}

void delete_node_by_node(b_tree* t, node* v) {
    if (v == NULL) {
```

```
        return;
    }
    node* i = v;

    node* l = v->left,* r = v->right;

    if (!(l == NULL)) {
        delete_node_by_node(t, l);
    }
    if (!(r == NULL)) {
        delete_node_by_node(t, r);
    }
    if (!(i->p == NULL)) {
        if (i->p->left == i) {
            i->p->left = NULL;
        } else {
            i->p->right = NULL;
        }
    }
    free(i);
    --t->last_id;
    --t->n;
    if (t->n == 0) {
        t->root = NULL;
    }
}

int64_t data_by_id(b_tree* const t, int64_t id) {
    node* v = node_by_id(t->root, id);
    if (v == NULL) {
        exception_exit("There is no node with given id in the b_tree\n")
    }
    return v->data;
}

int8_t node_is_in_b_tree(node* const root, node* const v) {
    if (v == NULL || root == NULL) {
        return 0;
    }
    if (root == v) {
        return 1;
    }
    return node_is_in_b_tree(root->left, v) \
        || node_is_in_b_tree(root->right, v);
}

void clear_tree(b_tree* t) {
    if (!is_empty_b_tree(t)) {
        delete_node_by_node(t, t->root);
        clear_imap(&(t->d));
        clear_imap(&(t->c));
    } else {
        t->n = 0;
        t->last_id = -1;
        t->c = create_empty_imap();
        t->d = create_empty_imap();
        t->root = NULL;
    }
}

void update_depth(b_tree* t) {
    node* i = t->root;
    if (i == NULL) {
        clear_tree(t);
        return;
    }
    clear_imap(&(t->d));
    __update_depth(&(t->d), t->root, 0);
}

void __update_depth(imap* d, node* const root, int64_t depth) {
    if (root == NULL) {
        return;
    }
    imap_add(d, root->id, depth);
    __update_depth(d, root->left, depth + 1);
    __update_depth(d, root->right, depth + 1);
}

void update_degree(b_tree* t) {
    node* i = t->root;
    if (i == NULL) {
        clear_tree(t);
        return;
    }
```

```
    clear_imap(&(t->c));
    __update_degree(&(t->c), t->root);
}

void __update_degree(imap* c, node* const root) {
    if (root == NULL) {
        return;
    }
    int8_t c_ = 0;
    if (!(root->left == NULL)) {
        ++c_;
        __update_degree(c, root->left);
    }
    if (!(root->right == NULL)) {
        ++c_;
        __update_degree(c, root->right);
    }
    imap_add(c, root->id, c_);
}

void print_b_tree(node* const root, uint8_t tab, SIDE side) {
    if (root == NULL) {
        return;
    }
    printf("%ld", root->id);
    for (uint8_t i = 0; i <= tab; ++i) {
        printf("\t");
        printf("|");
    }
    if (side == N_LEFT) {
        printf("L: ");
    } else if (side == N_RIGHT) {
        printf("R: ");
    } else {
        printf("Root: ");
    }
    printf("%ld\n", root->data);
    print_b_tree(root->left, tab + 1, N_LEFT);
    print_b_tree(root->right, tab + 1, N_RIGHT);
    //printf("\n");
}

int8_t is_B_tree(b_tree* const t) {
    if (is_empty_b_tree(t)) {
        return 1;
    }
    b_tree t_ = create_empty_b_tree();
    copy_b_tree(&t_, t);
    update_degree(&t_);
    imap_item* iterator = t_.c.begin;
    while (! (iterator == t_.c.back)) {
        if (iterator->value % 2) {
            return 0;
        }
        iterator = iterator->next;
    }
    if (iterator->value % 2) {
        return 0;
    }
    return 1;
}bulat@bulat-Swift-SF314-58:~/Studying/prprm/l/l23$ cat main.c
#include <inttypes.h>
#include <stdio.h>
#include <stdlib.h>
#include "string.h"
#include "tree.h"

typedef enum COMMAND {
    ADD_NODE,    // add node $id$ $value$ $side$ or add root $value$
    DELETE_NODE, // delete $id$
    PRINT_TREE,  // print
    // Calculating function checks if tree is a B-tree
    CALC_FUNC,   // calc
    EXIT,        // exit
    PASS,        // pass or Enter
    HISTORY      // history
} COMMAND;

string* split(string* const s, string* const sep) {
    string* list = (string*)malloc(10 * sizeof(string));
    for (int8_t i = 0; i < 10; ++i) {
        list[i] = not_allocated_string();
    }
    string s_; construct_empty(&s_);
    char ch;
```

```
    uint8_t size = 0;
    int8_t skip = 0;
    for (uint64_t i = 0; i < s->last_char; ++i) {
        ch = *at(s, i);
        for (uint64_t j = 0; j < sep->last_char; ++j) {
            if (ch == *at(sep, j)) {
                construct_from_s(&list[size], s_);
                ++size;
                destruct(&s_);
                construct_empty(&s_);
                skip = 1;
                break;
            }
        }
        if (skip) {
            skip = 0;
            continue;
        }
        add_char(&s_, ch);
    }
    return list;
}

void clear_split(string* list) {
    for (uint64_t i = 0; i < 10; ++i) {
        if (list[i].s == NULL)
            break;
        destruct(&list[i]);
    }
}

int main(int64_t argc, char** argv) {

    string history, ll, sep;
    construct_empty(&history); construct_empty(&ll); construct_empty(&sep);

    add_char(&sep, ' ');
    add_char(&sep, '\n');

    printf("Please, input command:\n");
    int64_t line = 1;

    b_tree t = create_empty_b_tree();
    char* pEnd;

    while (1) {
        printf("%ld: ", line);
        construct_empty(&ll);
        read_line(&ll);
        if (ll.last_char == 0) {
            destruct(&ll);
            continue;
        }
        add_string(&history, ll);
        string* list = split(&ll, &sep);
        if        (equal_charp(&list[0], "add")) {
            if (equal_charp(&list[1], "root")) {
                int64_t val = atoll(list[2].s);
                init_b_tree(&t, val);
            } else if (equal_charp(&list[1], "node")) {
                if (is_empty_b_tree(&t)) {
                    printf("Tree is empty. \nAdd root with command: add root <value>\n");
                } else {
                    int64_t id = atoll(list[2].s);
                    int64_t val = atoll(list[3].s);
                    SIDE s;
                    if (equal_charp(&list[4], "left"))
                        s = N_LEFT;
                    else if (equal_charp(&list[4], "right"))
                        s = N_RIGHT;
                    if (node_by_id(t.root, id) == NULL) {
                        printf("There is no node in tree with id %ld\n", id);
                    } else {
                        add_node_by_id(&t, id, val, s);
                    }
                }

            } else {
                printf("Wrong syntax\n");
            }
        } else if (equal_charp(&list[0], "delete")) {
            int64_t id = atoll(list[1].s);
            delete_node_by_id(&t, id);
        } else if (equal_charp(&list[0], "print")) {
            print_b_tree(t.root, 0, N_ROOT);
```

```
        } else if (equal_charp(&list[0], "calc")) {
            printf("This tree is%s%c", (is_B_tree(&t) ? " a B-tree" : " not a B-tree"), '\n');
        } else if (equal_charp(&list[0], "exit")) {
            break;
        } else if (equal_charp(&list[0], "history")) {
            print(history);
        } else {
            printf("Wrong command\n");
        }
        clear_split(list);
        destruct(&ll);
       ++line;
    }

    destruct(&history);
    destruct(&ll);
    destruct(&sep);

    clear_tree(&t);

    return 0;
}bulat@bulat-Swift-SF314-58:~/Studying/prprm/l/l23$ gcc -g imap.c string.c tree.c main.c -o main
bulat@bulat-Swift-SF314-58:~/Studying/prprm/l/l23$ ./main
Please, input command:
1: add root 0
2: add node 0 1 left
3: add node 0 2 right
4: add node 1 3 left
5: add node 1 4 right
6: print
0 |Root: 0
1 | |L: 1
3 | | |L: 3
4 | | |R: 4
2 | |R: 2
7: calc
This tree is a B-tree
8: add node 2 5 left
9: print
0 |Root: 0
1 | |L: 1
3 | | |L: 3
4 | | |R: 4
2 | |R: 2
5 | | |L: 5
10: calc
This tree is not a B-tree
11: delte 2
Wrong command
12: delete 2
13: print
0 |Root: 0
1 | |L: 1
3 | | |L: 3
4 | | |R: 4
14: calc
This tree is not a B-tree
15: delete 1
16: print
0 |Root: 0
17: calc
This tree is a B-tree
18: delte 0
Wrong command
19: delete 0
20: print
21: add node 0 1
Tree is empty.
Add root with command: add root <value>
22: add root -10
23: print
0 |Root: -10
24: add node 0 1 124
25: add node 9 0 12235
There is no node in tree with id 9
26: print
0 |Root: -10
1 | |L: 1
This tree is not a B-tree
27: calc
28: add node 0 2 right
29: print
0 |Root: -10
1 | |L: 1
2 | |R: 2
```

```
30: calc
This tree is a B-tree
31: exit
bulat@bulat-Swift-SF314-58:~/Studying/prprm/l/l23$ exit
exit
Script done.
```

9. **Дневник отладки** должен содержать дату и время сеансов отладки и основные события (ошибки в сценарии и программе, нестандартные ситуации) и краткие комментарии к ним. В дневнике отладки приводятся сведения об использовании ЭВМ, существенном участии преподавателя и других лиц в написании и отладке программы.

№	Лаб. или дом.	Дата	Время	Событие	Действие по исправлению	Примечание

10. **Замечания автора** по существу работы: _____

_

_

_____

11. **Выводы:** В ходе этой лабораторной работы я получил опыт реализации некоторых структур данных, работы с ними и парсинга текста. _____

_

_

_

Недочёты при выполнении задания могут быть устранены следующим образом: _

_

_

_

Подпись студента _____