Министерство образования Республики Беларусь

Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Кафедра вычислительных методов и программирования

Отчет по лабораторной работе №5
Нелинейные списки
Вариант 11

Выполнил:
студент 1 курса
группы № 348602
Трошкин Дмитрий Сергеевич

Проверил:
Матюшкин Светослав Иванович

Минск 2023

1 НЕЛИНЕЙНЫЕ СПИСКИ

Цель работы: изучить алгоритмы обработки данных с использованием нелинейных структур в виде дерева.

1.1 Условие

Разработать проект для работы с деревом поиска, содержащий следующие обработчики, которые должны: – ввести информацию (желательно, используя StringGrid), состоящую из целочисленного ключа и строки текста (например, номер паспорта и ФИО); – записать информацию в дерево поиска; – сбалансировать дерево поиска; – добавить в дерево поиска новую запись; – по заданному ключу найти информацию и отобразить ее; – удалить из дерева поиска информацию с заданным ключом; – распечатать информацию прямым, обратным обходом и в порядке возрастания ключа; – решить одну из следующих задач. Решение поставленной задачи оформить в виде блок-схемы.

1.2 Исходный код

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

const int max_length;
char tree_draw_helper[] = "/ \\";

typedef struct Tree {
    int value;
    char *name;
    struct Tree *left, *right;
} tree;

typedef struct NameValue {
    int val;
    char *name;
} nameVal;

typedef int (*tree_cmp) (const tree* node, const int value, const char *name, const tree* ptr);

tree *list(const int value, const char *name)
{
    tree *t = malloc(sizeof(tree));
    t->value = value;
    t->left = t->right = NULL;
    t->name = strdup(name);
    return t;
}
```

```
int isExisting(const tree *root, const int key)
{
    return (root ? (key == root->value ? 1 : isExisting(root->left, key)
            + isExisting(root->right, key)) : 0);
}
void add_list(tree **root, const int value, const char *name)
{
    if(!(*root)) {
        *root = list(value, name);
        return;
    }
    if(isExisting(*root, value)) {
        printf("Duplicated value\n");
        return;
    }
    if(((*root)->value) > value)
        add_list(&((*root)->left), value, name);
    else
        add_list(&((*root)->right), value, name);
}

void concat(char *target, const char *source)
{
    if(!target || !source) return;
    for(; *target; target++);
    for(; *source; source++, target++) *target = *source;
    *target = '\0';
}

int digits_count(int val)
{
    int i = 1;
    for(val /= 10; val; i++, val /= 10);
    return i;
}

int get_tree_length(const tree *root)
{
    int left_len, right_len;
    if(!root) return 0;

    left_len = get_tree_length(root->left);
    right_len = get_tree_length(root->right);

    return 1 + ((left_len > right_len) ? left_len : right_len);
}

int twoinpower(int power)
{
    return 1 << power;
}

int tree_to_strs(tree *root, char **strs, int dist_to_peek,
                 int direction, const char *separator, const int max_len)
{
    char *tmp;
    int d_count, padding_left = 0, padding_right = 0, width = max_len;
    if(root) {
        tmp = malloc( twoinpower(dist_to_peek) * max_len * sizeof(char));
```

```
        d_count = digits_count(root->value);
        padding_left = tree_to_strs(root->left, strs+1,
                       dist_to_peek - 1, direction, separator, max_len);
        padding_right = tree_to_strs(root->right, strs+1,
                                     dist_to_peek - 1, direction, separator, max_len);
        width = padding_left + padding_right - d_count - 2;
        sprintf(tmp,
                    "%*s%c%i%c%*s",
                    (width / 2) + (width % 2), "",
                    tree_draw_helper[1 - direction * !(!root->left)],
                    root->value,
                    tree_draw_helper[1 + direction * !(!root->right)],
                    (width / 2), ""
                );
        concat(*strs, tmp);
        free(tmp);

        return padding_left + padding_right;
    }
    if(dist_to_peek > 1) width = tree_to_strs(NULL, strs+1,
                                    dist_to_peek - 1, direction, separator, max_len);
    if(dist_to_peek > 0) {
        concat(*strs, separator);
    }
    return width;
}

int get_max(tree *root)
{
    if(!root) return 0;
    return root->right ? get_max(root->right) : root->value;
}

void view_tree(tree *root, int direction)
{
    int max_num_len = digits_count(get_max(root)) + 2;
    int dist_to_peek = get_tree_length(root) + 1;
    char **strs = malloc((dist_to_peek) * sizeof(char*));
    int str_size = twoinpower(dist_to_peek) * max_num_len;
    for(int i = 0; i < dist_to_peek; i++) {
        strs[i] = malloc(str_size * sizeof(char));
        strs[i][0] = 0;
    }
    char *separ = malloc((max_num_len + 1) * sizeof(char));
    for(int i = 0; i < max_num_len; i++)
        separ[i] = ' ';

    separ[max_num_len] = '\0';

    tree_to_strs(root, strs, dist_to_peek, direction, separ, max_num_len);

    for(int i = (direction > 0) ? 0 : (dist_to_peek - 2);
        (direction > 0) ? (i < dist_to_peek - 1) : (i >= 0); i += direction) {
        puts(strs[i]);
        free(strs[i]);
    }
    free(strs);
}
```

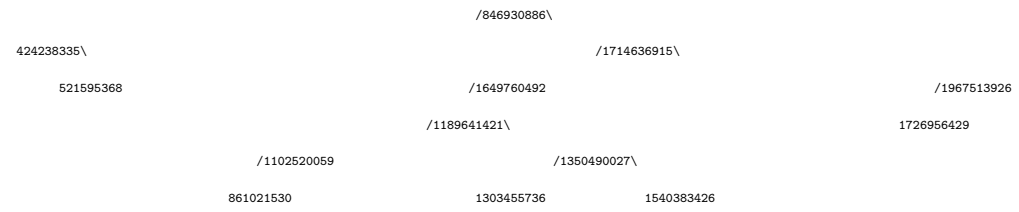
```
tree *del_by_key(tree *root, const int key)
{
    tree *del, *del_prev, *r, *r_prev;
    del = root;
    del_prev = NULL;

    while(del != NULL && del->value != key) {
        del_prev = del;
        del = (del->value > key) ? del->left : del->right;
    }

    if(del == NULL) {
        puts("Key not found!");
        return root;
    }

    if(del->left == NULL) r = del->right;
    else if(del->right == NULL) r = del->left;
    else {
        r_prev = del;
        r = del->left;
        while(r->right != NULL)
            r_prev = r, r = r->right;

        if(r_prev == del) r->right = del->right;
        else {
            r->right = del->right;
            r_prev->right = r->left;
            r->left = r_prev;
        }
    }

    if(del == root) root = r;
    else if (del->value < del_prev->value) del_prev->left = r;
    else del_prev->right = r;

    free(del);
    return root;
}

void del_tree(tree *root)
{
    if(root) {
        del_tree(root->left);
        del_tree(root->right);
        free(root);
    }
}

tree *sorted_arr_to_tree(nameVal *arr, int len)
{
    int center = len / 2;
    if(len <= 0) return NULL;
    tree *t = list(arr[center].val, arr[center].name);
    if(len > 1) {
        t->left = sorted_arr_to_tree(arr, center);
        t->right = sorted_arr_to_tree(arr + center + 1, len - center - 1);
    }
    return t;
```

```
}

tree *find_node_by_key(tree *root, int key)
{
    if(!root) return NULL;
    if(root->value == key) return root;
    return (root->value > key) ?
            find_node_by_key(root->left, key) :
            find_node_by_key(root->right, key);
}

void print_tree_straight(tree *root)
{
    if(root) {
        printf("%i: %s\n", root->value, root->name);
        print_tree_straight(root->left);
        print_tree_straight(root->right);
    }
}

void print_tree_reverse(tree *root)
{
    if(root) {
        print_tree_reverse(root->left);
        print_tree_reverse(root->right);
        printf("%i: %s\n", root->value, root->name);
    }
}
void print_tree_middle(tree *root)
{
    if(root) {
        print_tree_middle(root->left);
        printf("%i: %s\n", root->value, root->name);
        print_tree_middle(root->right);
    }
}

/* tree_cmp must return
 * -1 root's val is less
 *  0 root's val is equal
 * +1 root's val is more
 *
 */

int compare_name_starts_with(const tree *root, const int value, const char *name, const tree *ptr)
{
    int i = 0;
    if(!root) return -2;
    while(name[i] && root->name[i] == name[i]) i++;
    return root->name[i] == name[i] ? 0 :
           root->name[i] > name[i] ? 1 : -1;

}


// Returns true if root->name contains name
int compare_name_lightly(const tree *root, const int value, const char *name, const tree *ptr)
{
    int i = 0;
```

```
    if(!root) return -2;
    while(name[i] && root->name[i] == name[i]) i++;
    return root->name[i] == name[i] ? 0 :
           root->name[i] > name[i] ? 1 : -1;

}

int check_starts_with_a(const tree *root, const int value, const char *name, const tree *ptr)
{
    return (root && root->name) ? (root->name[0] == 'a' || root->name[0] == 'A') : 0;
}

int tree_counter(const tree *root, const tree_cmp cmp, const int value,
                 const char *name, const tree *ptr, const int mode)
{
    if(!root) return 0;

    return (cmp(root, value, name, ptr) == mode) +
            tree_counter(root->left, cmp, value, name, ptr, mode) +
            tree_counter(root->right, cmp, value, name, ptr, mode);
}

void view_filtered(const tree *root, const tree_cmp cmp, const int value,
                   const char *name, const tree *ptr, const int mode)
{
    if(!root) return;

    view_filtered(root->left, cmp, value, name, ptr, mode);
    if(cmp(root, value, name, ptr)) printf("%i: %s\n", root->value, root->name);
    view_filtered(root->right, cmp, value, name, ptr, mode);
}

int get_count(const tree *root)
{
    return root ? 1 + get_count(root->left) + get_count(root->right) : 0;
}

int tree_to_arr(const tree *root, nameVal *arr)
{
    int index = 0;
    if(!root) return 0;

    index = tree_to_arr(root->left, arr);
    arr[index].val = root->value;
    arr[index].name = root->name;
    index += tree_to_arr(root->right, arr + index + 1);
    return index + 1;
}

int main()
{
    tree *tmp = NULL, *root = NULL;

    const char names[30][15] = {
        "Andrew",
        "Aleksey",
        "Akakiy",
        "Boris",
        "Vicror",
```

```
        "Georgiy",
        "Dmitriy",
        "Evgeniy",
        "Zhorik",
        "Zina",
        "Irina",
        "Katya",
        "Lena",
        "Michail",
        "aboba"
    };

    int count = 13;
    nameVal arr[64];

    /*for(int i = 0; i < count; i++)
        arr[i].val = i, arr[i].name = names[rand() % 15];
    root = sorted_arr_to_tree(arr, count);
    */
    for(int i = 0; i < count; i++)
        add_list(&root, rand(), names[rand() % 15]);

    view_tree(root, 1);

    puts("\nПрямой обход"); print_tree_straight(root);
    puts("\nОбратный обход"); print_tree_reverse(root);
    puts("\nВ порядке возростания ключа"); print_tree_middle(root);

    printf("Amount of names starting with 'a': %i\n",
            tree_counter(root, check_starts_with_a, 0, 0, 0, 1));

    view_filtered(root, check_starts_with_a, 0, 0, 0, 1);

    int c = get_count(root);
    nameVal *array = malloc(c * sizeof(nameVal));

    tree_to_arr(root, array);


    tmp = sorted_arr_to_tree(array, c);
    view_tree(tmp, -1);
    puts("\n");
    view_tree(tmp, 1);

    del_tree(root);
    del_tree(tmp);
}
```

1.3 Пример



```
Прямой обход
846930886: Michail
424238335: Zhorik
521595368: Katya
1714636915: Lena
1649760492: Aleksey
1189641421: Zina
1102520059: Georgiy
861021530: Akakiy
1350490027: Akakiy
1303455736: Evgeniy
1540383426: Andrew
1967513926: Zhorik
1726956429: Lena

Обратный обход
521595368: Katya
424238335: Zhorik
861021530: Akakiy
1102520059: Georgiy
1303455736: Evgeniy
1540383426: Andrew
1350490027: Akakiy
1189641421: Zina
1649760492: Aleksey
1726956429: Lena
1967513926: Zhorik
1714636915: Lena
846930886: Michail

В порядке возростания ключа
424238335: Zhorik
521595368: Katya
846930886: Michail
861021530: Akakiy
1102520059: Georgiy
1189641421: Zina
1303455736: Evgeniy
1350490027: Akakiy
1540383426: Andrew
1649760492: Aleksey
1714636915: Lena
1726956429: Lena
1967513926: Zhorik
Amount of names starting with 'a': 4
861021530: Akakiy
1350490027: Akakiy
1540383426: Andrew
1649760492: Aleksey
        424238335               846930886             1102520059                      1350490027             1649760492             1726956429
                  \521595368/                           \1189641421                           \1540383426/                           \1967513926
```

\861021530/ \1714636915/

\1303455736/

/1303455736\

/861021530\ /1714636915\

/521595368\ /1189641421 /1540383426\ /1967513926

424238335 846930886 1102520059 1350490027 1649760492 1726956429