ГУАП

КАФЕДРА № 34

ОТЧЕТ
ЗАЩИЩЕН С ОЦЕНКОЙ

ПРЕПОДАВАТЕЛЬ

доц., канд. техн. наук		К. А. Жиданов
должность, уч. степень, звание	подпись, дата	инициалы, фамилия

ОТЧЕТ О ЛАБОРАТОРНОЙ РАБОТЕ

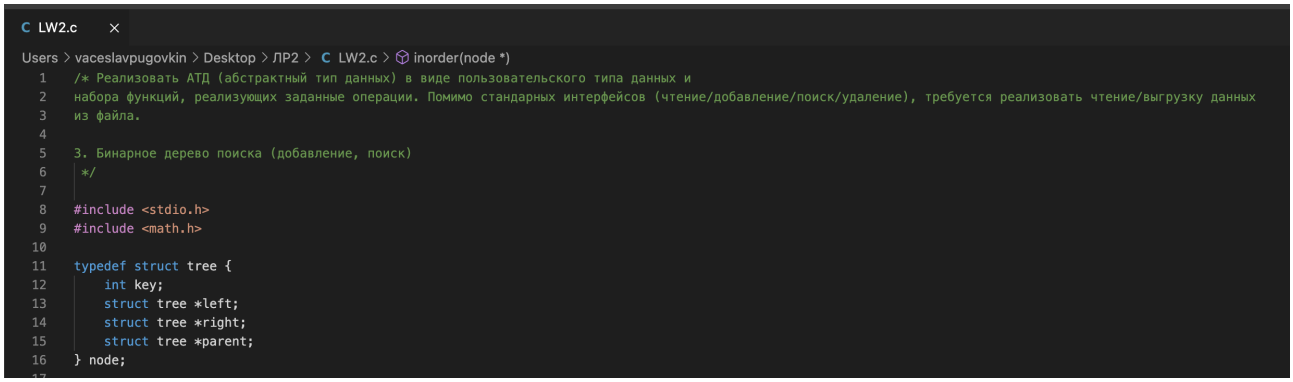
Реализация АТД. Бинарное дерево.

по курсу: Языки программирования

РАБОТУ ВЫПОЛНИЛ

СТУДЕНТ ГР. №	3145	24.05.22	В. В. Пуговкин
		подпись, дата	инициалы, фамилия

Санкт-Петербург 2022



- Используем **typedef** для создания нового типа, чтобы в дальнейшем не писать слово struct.
- **int key** — ключ, может быть любого типа.
- **struct tree *left** — указатель на левое поддерево.
- **struct tree *right** — указатель на правое поддерево.
- **struct tree *parent** — указатель на родителя.
- **node** — название структуры.

```
node *create(node *root, int key) {
    FILE *S;
    char name[] = "text.txt";
    if ((fp = fopen(name, "r")) == NULL) {
    printf("Не удалось открыть файл");
    getchar();
    return 0;
    }
// Выделение памяти под корень дерева
    node *tmp = malloc(sizeof(node));
// Присваивание значения ключу
    tmp -> key = key;
// Присваивание указателю на родителя значения NULL
    tmp -> parent = NULL;
// Присваивание указателю на левое и правое поддерево значения NULL
    tmp -> left = tmp -> right = NULL;
    root = tmp;

    fprintf(S, "%d", root);
    fclose(S);
    return root;

}
```

Мы инициализируем дерево отдельной функцией для того, чтобы облегчить процесс добавления узлов в дерево. Другими словами, мы создаем корень бинарного дерева поиска. + осуществляем выгрузку в файл.

```
node *add(node *root, int key)
{
    node *root2 = root, *root3 = NULL;
// Выделение памяти под узел дерева
    node *tmp = malloc(sizeof(node));
// Присваивание значения ключу
    tmp -> key = key;
/* Поиск нужной позиции для вставки (руководствуемся правилом
вставки элементов, см. начало статьи, пункт 3) */
    while (root2 != NULL)
    {
        root3 = root2;
        if (key < root2 -> key)
            root2 = root2 -> left;
        else
            root2 = root2 -> right;
    }
/* Присваивание указателю на родителя значения указателя root3
(указатель root3 был найден выше) */
    tmp -> parent = root3;
// Присваивание указателю на левое и правое поддерево значения NULL
    tmp -> left = NULL;
    tmp -> right = NULL;
/* Вставляем узел в дерево (руководствуемся правилом
вставки элементов, см. начало статьи, пункт 3) */
    if (key < root3 -> key) root3 -> left = tmp;
    else root3 -> right = tmp;
    return root;
}
```

- tmp -> left и tmp -> right имеют значение NULL, так как указатель tmp расположен в конце дерева.
- Указатель root2 использовался для того, чтобы сохранить адрес на родителя вставляемого узла.
- Мы не проверяем дерево на пустоту, так как ранее дерево был инициализировано (имеется корень).

```
node *search(node * root, int key)
{
// Если дерево пусто или ключ корня равен искомому ключу, то возвращается указатель на корень
    if ((root == NULL) || (root -> key == key))
        return root;
// Поиск нужного узла
    if (key < root -> key)
        return search(root -> left, key);
    else return search(root -> right, key);
}

// Минимальный элемент дерева
node *min(node *root)
{
    node *l = root;
    while (l -> left != NULL)
        l = l -> left;
    return l;
}

// Максимальный элемент дерева
node *max(node *root)
{
    node *r = root;
    while (r -> right != NULL)
        r = r -> right;
    return r;
}
```

Данная функция рекурсивная, поэтому комментарий «Если дерево пусто или ключ корня равен искомому ключу, то возвращается указатель на корень» является не совсем верным, потому что root указывает на корень только во время первой итерации, далее root ссылается на другие узлы дерева, но из-за рекурсивности функции условие *if ((root == NULL) || (root -> key == key))* будет проверяться всегда.

```
node *succ(node *root)
{
    node *p = root, *l = NULL;
// Если есть правое поддерево, то ищем минимальный элемент в этом поддереве
    if (p -> right != NULL)
        return min(p -> right);
/* Правое дерево пусто, идем по родителям до тех пор,
пока не найдем родителя, для которого наше поддерево левое */
    l = p -> parent;
    while ((l != NULL) && (p == l -> right))
    {
        p = l;
        l = l -> parent;
    }
    return l;
}
```

```
node *delete(node *root, int key)
{
// Поиск удаляемого узла по ключу
    node *p = root, *l = NULL, *m = NULL;
    l = search(root, key);
// 1 случай
    if ((l -> left == NULL) && (l -> right == NULL))
    {
        m = l -> parent;
        if (l == m -> right) m -> right = NULL;
        else m -> left = NULL;
        free(l);
    }
// 2 случай, 1 вариант – поддерево справа
    if ((l -> left == NULL) && (l -> right != NULL))
    {
        m = l -> parent;
        if (l == m -> right) m -> right = l -> right;
        else m -> left = l -> right;
        free(l);
    }
// 2 случай, 2 вариант – поддерево слева
    if ((l -> left != NULL) && (l -> right == NULL))
    {
        m = l -> parent;
        if (l == m -> right) m -> right = l -> left;
        else m -> left = l -> left;
        free(l);
    }
// 3 случай
    if ((l -> left != NULL) && (l -> right != NULL))
    {
        m = succ(l);
        l -> key = m -> key;
        if (m -> right == NULL)
            m -> parent -> left = NULL;
        else m -> parent -> left = m -> right;
        free(m);
    }
    return root;
}
```

Рассмотрим самый простой случай: у удаляемого узла нет левого и правого поддерева. В данной ситуации мы просто удаляем данный лист (узел).

У удаляемого узла одно поддерево. В данной ситуации мы просто удаляем данный узел, а на его место ставим поддерево.

Самый сложный случай: у удаляемого узла существуют оба поддерева. В данной ситуации необходимо сначала найти следующий за удаляемым элемент, а потом его поставить на место удаляемого элемента.

```
// Обход дерева в симметричном порядке, будет напечатано D B A E G C H F J
void inorder(node *root) {
    FILE *S;
    char name[] = "text.txt";
    int y;
    if ((fp = fopen(name, "r")) == NULL) {
        printf("Не удалось открыть файл");
        getchar();
        return 0;
    }

    if (root == NULL)
        return;
    inorder(root -> left);
    if (root -> key)
        printf("%d ", root -> key);
    inorder(root -> right);

    fscanf(S, "%d", &y);
    fprint(y)
    fclose(S);

}
```