**Теоретическое описание решения поставленной задачи**

Теоретическое определение бинарного дерева:

**Бинарное дерево или двоичное дерево — это дерево, в котором у каждого из его узлов не более двух дочерних узлов. При этом каждый дочерний узел тоже представляет собой бинарное дерево.**

**Благодаря тому, что дочерних узлов всегда не больше двух, их называют правый и левый дочерние узлы.**

**Есть завершенное и полное деревья. Для бинарных деревьев они приобретают следующий вид:**

**Завершенное бинарное дерево — это бинарное дерево, в котором каждый уровень, кроме последнего, полностью заполнен, а заполнение последнего уровня производится слева направо**

**Полное бинарное дерево — это бинарное дерево, в котором у каждого узла ноль или два дочерних узла.**

**Пример бинарного дерева:**

Изображение выглядит как круг, диаграмма, зарисовка, рисунок

Автоматически созданное описание

В данном примере бинарное дерево содержит числа от 2 до 30. Корень дерева - узел со значением 10. Левый потомок корня имеет значение 3, а его правый потомок - значение 12. Узлы дальше разветвляются, так что у левого потомка 3 есть потомки 2 и 4, а у правого потомка 12 - потомки 11 и 30.

Бинарное дерево имеет несколько основных операций, которые позволяют работать с ним:

Вставка – добавление нового узла в дерево. Новый узел может быть добавлен как левым, так и правым потомком существующего узла, в зависимости от условий, которые задаются при выполнении операции вставки.

Удаление – удаление узла из дерева. При удалении узла, его потомки (если они существуют) становятся потомками его родителя.

Поиск – поиск узла в дереве. Поиск может быть реализован как рекурсивным, так и итеративным способом.

Обход – обход дерева с целью выполнения определенных операций над его узлами. В зависимости от порядка обхода, результат операции может быть различным.

Бинарные деревья широко применяются в алгоритмах, таких как поиск, сортировка и обход графа. В различных задачах бинарное дерево может быть использовано для эффективного хранения и обработки данных.

Одним из наиболее часто используемых типов бинарного дерева является двоичное дерево поиска (Binary Search Tree, BST). В этом типе деревьев каждый узел содержит значение, которое является уникальным ключом для поиска и сортировки элементов. Узлы в BST располагаются таким образом, чтобы ключи слева от узла были меньше его значения, а справа – больше.

Бинарные деревья являются эффективными структурами данных, так как время выполнения операций (вставка, удаление, поиск) в них обычно пропорционально высоте дерева, а не числу узлов. Однако, в худшем случае, дерево может быть несбалансированным и иметь высоту, равную числу узлов, что может существенно снизить эффективность операций.

В заключении, бинарное дерево – это важная структура данных, которая находит применение во многих областях программирования и алгоритмов. Знание работы с ним позволяет разработчикам эффективно решать различные задачи, связанные с хранением и обработкой данных.

**Алгоритм реализации поставленной задачи**

Для выполнения поставленной задачи необходимо разработать алгоритм по ее реализации. Данный алгоритм состоит из следующих пунктов:

1. Описание класса элемента бинарного дерева;
2. Описание класса бинарного дерева;
3. Реализация методов поиска, добавления и вывода бинарного дерева;
4. Реализация функции поиска ближайших статей с большим номером со следующими условиями:
   1. Описание функции поиска заданной статьи;
   2. Проверка условий наличия статьи-родителя и сравнение значений текущей статьи и статьи родителя;
   3. Проверка наличия статьи-родителя и наличия правого потомка;
   4. Сравнение значений текущей статьи с правым потомком, либо сравнение значений текущей статьи со значением статьи-родителя и проверка наличия правого потомка;
   5. Проверка наличия правого потомка и сравнение значения текущей статьи со значением статьи-родителя;
   6. Описание цикла While с условием наличия статьи-родителя и сравнения значений статьи-родителя с заданным ключом;
5. Заполнение дерева значениями;
6. Вызов функции по нахождению статьи;
7. Вывод полученного значения.

**Листинг кода программы, реализующей поставленную задачу на языке C++**

#include <iostream>

#include <iomanip>

#include <windows.h>

#include <vector>

#include <string>

using namespace std;

class Item {

public:

    int data;

    Item\* parent;

    Item\* left\_child;

    Item\* right\_child;

    Item(int val) {

        parent = NULL;

        left\_child = NULL;

        right\_child = NULL;

        data = val;

    }

};

class BinaryTree {

private:

    Item\* root;

    int TreeSize;

    void print\_tree(Item\*);

    void delete\_tree(Item\*);

public:

    BinaryTree(int);

    ~BinaryTree();

    void print();

    void print\_tree(Item\*, int);

    void insert(int);

    void GreenFunc(int);

};

BinaryTree::BinaryTree(int key)

{

    root = new Item(key);

    TreeSize = 1;

}

BinaryTree::~BinaryTree()

{

    delete(root);

}

void BinaryTree::print()

{

    print\_tree(root);

    cout << endl;

}

void BinaryTree::print\_tree(Item\* curr)

{

    if (curr)

    {

        print\_tree(curr->left\_child);

        cout << " data - " << curr->data << " ";

        print\_tree(curr->right\_child);

    }

}

void BinaryTree::insert(int key)

{

    Item\* curr = root;

    while (curr && curr->data != key)

    {

        if (curr->data > key && curr->left\_child == NULL)

        {

            curr->left\_child = new Item(key);

            curr->left\_child->parent = curr;

            ++TreeSize;

            return;

        }

        if (curr->data < key && curr->right\_child == NULL)

        {

            curr->right\_child = new Item(key);

            curr->right\_child->parent = curr;

            ++TreeSize;

            return;

        }

        if (curr->data > key) {

            curr->left\_child->parent = curr;

            curr = curr->left\_child;

        }

        else {

            curr->right\_child->parent = curr;

            curr = curr->right\_child;

        }

    }

}

void BinaryTree::GreenFunc(int key) {

    Item\* curr = root;

    while (curr && curr->data != key)

    {

        if (curr->data > key) {

            curr->left\_child->parent = curr;

            curr = curr->left\_child;

        }

        else {

            curr->right\_child->parent = curr;

            curr = curr->right\_child;

        }

    }

    if (curr->parent != NULL && curr->data < curr->parent->data  ) {

        cout << curr->parent->data << endl;

    }

    else if (curr->parent == nullptr && curr->right\_child) {

        cout << curr->right\_child->data << endl;

    }

    else if (curr->right\_child > curr || curr->data > curr->parent->data && curr->right\_child) {

        cout << curr->right\_child->data << endl;

    }

    else if (!curr->right\_child && curr->data > curr->parent->data ) {

            while (curr->parent != nullptr && curr->parent->data < key ) {

                if (curr->parent) {

                    curr = curr->parent;

                }

            }

            if (curr->parent == nullptr) {

                cout << "This item do not have biggest number" << endl;

            }

            else {

                cout << curr->parent->data << endl;;

            }

    }

    else {

        cout << "This item do not have biggest number" << endl;;

    }

}

int main() {

    int insert = 0;

    BinaryTree Tree1(10);

    Tree1.insert(3);

    Tree1.insert(2);

    Tree1.insert(12);

    Tree1.insert(4);

    Tree1.insert(5);

    Tree1.insert(11);

    Tree1.insert(30);

    cout << "Please enter the number for search:" << endl;

    cin >> insert;

    cout << endl;

    cout << "Nearest item with bigger number - ";

    Tree1.GreenFunc(insert);

    system("PAUSE");

}