**Лабораторная работа №3.**

1. **Общая постановка задачи:**

**1. Реализовать структуру данных бинарное дерево поиска. Для этого создать шаблон класса BinarySearchTree**

***1.1 Для описания узла дерева используйте тип Node, в котором поля:***  
 • key\_ - значение ключа узла,  
 • left\_ - указатель на левое поддерево,  
 • right\_ - указатель на правое поддерево,  
 • p\_ - указатель на родителя (может не использоваться).  
Тип Node может использоваться только в классе BinarySearchTree

Класс BinarySearchTree должен содержать поле root\_ - указатель на корневой узел.

***1.2 В классе должны быть:***

• конструктор по умолчанию, создающий пустое дерево,  
• конструктор перемещения,  
• оператор перемещающего присваивания,  
• деструктор.

Конструктор копирования и оператор присваивания (с копированием) должны быть  
запрещены.

***1.3 В классе должны быть методы:***

1. поиска по ключу (итеративный)  
2. вставки нового элемента в дерево (итеративный)  
3. удаления элемента из дерева (итеративный)  
4. печати строкового изображения дерева  
5. определения количества узлов дерева (рекурсивный)  
6. определения высоты дерева (рекурсивный)  
7. инфиксного обхода дерева (итеративный)  
8. инфиксного обхода дерева (рекурсивный)  
9. итеративный метод обхода двоичного дерева по уровням (в ширину). В реализации  
использовать класс очередь  
10. определения, являются ли два дерева похожими. Похожими будем называть деревья поиска, содержащие одинаковые наборы ключей. Рекомендация: параллельно обходить инфиксным обходом сравниваемые деревья  
11. определения, есть одинаковые ключи в двух деревьях поиска. Рекомендация: параллельно обходить инфиксным обходом сравниваемые деревья

***Два набора методов:***  
1. private для работы с узлами – для разработчика класса  
2. public работы со значениями (ключами) – для пользователя  
Методы могут быть перегруженными, т. е. можно использовать одно и то же имя для  
private и public методов.

1. **Таблица с детальными требованиями и тест планом**

|  |  |  |  |
| --- | --- | --- | --- |
| Требования | Детальные требования | Данные | Ожидаемый результат |
| 1. **Тест IterativeSearch** | | | |
| Если новый элемент вставлен, возвращается true, false, если элемент уже существует (метод insert). Если искомый элемент найден, метод iterativeSearch возвращает true, иначе false. Также метод print осуществляет печать строкового изображения дерева в выходной поток out, используя скобки, чтобы показать структуру дерева. | void searchTest()  {  std::cout << "---ТЕСТ МЕТОДА IterativeSearch---\n";  BinarySearchTree<int> bst;  std::cout << bst.insert(5) << "\n";  std::cout << bst.insert(3) << "\n";  std::cout << bst.insert(7) << "\n";  std::cout << bst.insert(2) << "\n";  std::cout << bst.insert(4) << "\n";  std::cout << bst.insert(6) << "\n";  std::cout << bst.insert(8) << "\n";  std::cout << bst.insert(8) << "\n";  bst.print(std::cout);    std::cout << "\n" << bst.iterativeSearch(2) << "\n";  std::cout << bst.iterativeSearch(10) << "\n";  } |  |  |
| 1. **Тест deleteKey** | | | |
| Метод удаляет элемент из дерева, не нарушая порядок элементов. Возвращает true, если элемент удален; false, если элемента не было. | void deleteTest()  {  std::cout << "\n---ТЕСТ МЕТОДА deleteKey---\n";  BinarySearchTree<int> bst;  bst.insert(5);  bst.insert(3);  bst.insert(7);  bst.insert(2);  bst.insert(4);  bst.insert(6);  bst.insert(8);  bst.print(std::cout);  std::cout << "\nУдаление несуществующего элемента\n";  std::cout << bst.deleteKey(10) << "\n";  std::cout << "\nЕсли узел не имеет потомков\n";  std::cout << bst.deleteKey(8) << "\n";  bst.print(std::cout);  std::cout << "\nЕсли узел имеет одного потомка - левого\n";  std::cout << bst.deleteKey(7) << "\n";  bst.print(std::cout);  std::cout << "\nЕсли узел имеет одного потомка - правого\n";  bst.deleteKey(2);  std::cout << bst.deleteKey(3) << "\n";  bst.print(std::cout);  std::cout << "\nЕсли узел имеет двух потомков\n";  BinarySearchTree<int> bst1;  bst1.insert(8);  bst1.insert(3);  bst1.insert(10);  bst1.insert(1);  bst1.insert(6);  bst1.insert(14);  bst1.insert(4);  bst1.insert(7);  bst1.insert(13);  bst1.print(std::cout);  std::cout << "\n" << bst1.deleteKey(3) << "\n";  bst1.print(std::cout);  std::cout << "\n";  } |  |  |
| 1. **Тест getCount и getHeight** | | | |
| getCount определяет количество узлов в дереве.  getHeight определяет высоту дерева. | void CountHeightTest()  {  std::cout << "\n---ТЕСТ МЕТОДОВ getCount и getHeight---\n";  BinarySearchTree<int> bst;  bst.insert(5);  bst.insert(3);  bst.insert(7);  bst.insert(2);  bst.insert(4);  bst.insert(6);  bst.insert(8);  bst.print(std::cout);  std::cout << "\nКоличество узлов дерева: " << bst.getCount();  std::cout << "\nВысота дерева: " << bst.getHeight() << "\n";  } |  |  |
| 1. **Тест InorderWalk (итеративный и рекурсивный)** | | | |
| Осуществляет инфиксный обход дерева (слева направо) | void InorderWalkTest()  {  std::cout << "\n---ТЕСТ МЕТОДОВ InorderWalk (итеративный и рекурсивный)---\n";  BinarySearchTree<int> bst;  bst.insert(8);  bst.insert(3);  bst.insert(10);  bst.insert(1);  bst.insert(6);  bst.insert(14);  bst.insert(4);  bst.insert(7);  bst.insert(13);  bst.print(std::cout);  std::cout << "\n";  bst.iterativeInorderWalk();  std::cout << "\n";  bst.inorderWalk();  } |  |  |
| 1. **Тест WalkByLevels** | | | |
| Осуществляет обход двоичного дерева по уровням (в ширину). В реализации используется класс очередь. | void WalkByLevelsTest()  {  std::cout << "\n\n---ТЕСТ МЕТОДА walkByLevels---\n";  BinarySearchTree<int> bst;  bst.insert(5);  bst.insert(3);  bst.insert(7);  bst.insert(2);  bst.insert(4);  bst.insert(6);  bst.insert(8);  bst.print(std::cout);  std::cout << "\n";  bst.walkByLevels();  } |  |  |
| 1. **Тест isSimilar** | | | |
| Определяет, являются ли два дерева похожими (деревья содержащие одинаковые наборы ключей). Реализация: параллельный обход инфиксным способом сравниваемых деревьев | void isSimilarTest()  {  std::cout << "\n\n---ТЕСТ МЕТОДА isSimilar---\n";  BinarySearchTree<int> bst1;  bst1.insert(10);  bst1.insert(7);  bst1.insert(15);  bst1.insert(5);  bst1.insert(8);  bst1.insert(11);  bst1.insert(17);  bst1.insert(20);  std::cout << "Дерево bst1: ";  bst1.print(std::cout);  std::cout << "\n";  BinarySearchTree<int> bst2;  bst2.insert(15);  bst2.insert(7);  bst2.insert(17);  bst2.insert(5);  bst2.insert(10);  bst2.insert(8);  bst2.insert(11);  bst2.insert(20);  std::cout << "Дерево bst2: ";  bst2.print(std::cout);  std::cout << "\n";  std::cout << bst1.isSimilar(bst2);  std::cout << "\nУдалим один ключ в bst2\n";  bst2.deleteKey(8);  std::cout << bst1.isSimilar(bst2);  } |  |  |
| 1. **Тест isIdenticalKey** | | | |
| Определяет, есть ли одинаковые ключи в двух деревьях поиска. Способом. Реализация: параллельный обход инфиксным способом сравниваемых деревьев | void isIdenticalKeyTest()  {  std::cout << "\n\n---ТЕСТ МЕТОДА isIdenticalKey---\n";  BinarySearchTree<int> bst1;  bst1.insert(5);  bst1.insert(3);  bst1.insert(7);  bst1.insert(2);  bst1.insert(4);  bst1.insert(6);  bst1.insert(8);  std::cout << "Дерево bst1: ";  bst1.print(std::cout);  std::cout << "\n";  BinarySearchTree<int> bst2;  bst2.insert(8);  bst2.insert(3);  bst2.insert(10);  bst2.insert(1);  bst2.insert(6);  bst2.insert(14);  bst2.insert(4);  bst2.insert(7);  bst2.insert(13);  std::cout << "Дерево bst2: ";  bst2.print(std::cout);  std::cout << "\n";  std::cout << bst1.isIdenticalKey(bst2);  std::cout << "\nУдалим все одинаковые ключи из bst2\n";  bst2.deleteKey(3);  bst2.deleteKey(7);  bst2.deleteKey(8);  bst2.deleteKey(6);  bst2.deleteKey(4);  std::cout << bst1.isIdenticalKey(bst2);  } |  |  |