**数据结构实验报告**

|  |  |  |  |
| --- | --- | --- | --- |
| 学号-姓名 |  | 实验时间 | 2023 年 11月24日 |
| 诚信声明 | 手写，签名 | | |
| 实验题目 | 二叉排序树的建立、查找、插入和删除运算 | | |
| 实验过程中遇到的主要问题 | 删除指定关键字，要处理不同情况下节点的替换和调整，包括叶子节点、只有左子树或右子树的节点，以及既有左子树又有右子树的节点，他们的操作各不相同，其中既有左子树又有右子树的节点较为复杂。 | | |
| 实验小结 | 这个实验使我对二叉排序树的建立、查找、插入和删除运算有了更深入的理解。二叉排序树可以快速地进行关键字的查找和插入操作。删除操作较为复杂，需要考虑不同情况下节点的替换和调整。中序遍历运算与普通二叉树相同。 | | |
| 数据结构  （自定义数据类型） | struct TreeNode {  int key;  TreeNode \*left;  TreeNode \*right;  };  class BinarySearchTree {  private:  TreeNode \*root; // 根结点  TreeNode \*insertRecursive(TreeNode \*node, int key); // 插入  bool searchRecursive(TreeNode \*node, int key); // 搜索  TreeNode \*removeRecursive(TreeNode \*node, int key); // 删除  void inorderTraversalRecursive(TreeNode \*node); // 中序遍历  int findPred(TreeNode \*node); // 找中序前驱  public:  void insert(int key); // 插入的包装方法  bool search(int key); // 搜索的包装方法  void remove(int key); // 删除的包装方法  void inorderTraversal(); // 中序遍历的包装方法  }; | | |
| 主要算法  （或算法说明） | 1. 查找：   * 从根节点开始，如果根节点为空，则返回空。 * 如果关键字等于根节点的关键字，则返回根节点。 * 如果关键字小于根节点的关键字，则递归地在左子树上查找；关键字大于根节点的关键字，则递归地在右子树上查找。   2. 插入：   * 如果根节点为空，则创建一个新节点，并将关键字赋值给新节点。 * 如果关键字小于根节点的关键字，则递归地在左子树上插入；关键字大于根节点的关键字，则递归地在右子树上插入。   3. 删除：   * 根节点为空，则返回空。 * 关键字小于根节点的关键字，则递归地在左子树上删除；关键字大于根节点的关键字，则递归地在右子树上删除。 * 如果关键字等于根节点的关键字，则进行删除操作： * 如果根节点没有左子树和右子树，则直接删除根节点。 * 如果根节点只有左子树或右子树，则将对应子树替换为根节点。 * 如果根节点既有左子树又有右子树，则找到中序遍历的前驱节点，将前驱节点的关键字替换到根节点，然后在左子树上递归地删除前驱节点。   4. 中序遍历：   * 如果根节点为空，则返回空。 * 递归地对左子树进行中序遍历。 * 输出根节点的关键字。 * 递归地对右子树进行中序遍历。 | | |

**数据结构实验报告**

|  |  |  |  |
| --- | --- | --- | --- |
| 学号-姓名 |  | 实验时间 | 2023 年 11月24日 |
| 诚信声明 | 手写，签名 | | |
| 实验题目 | 哈希表的设计和应用 | | |
| 实验过程中遇到的主要问题 | 哈希函数是实现哈希表关键。哈希函数的设计应尽量使关键字在哈希表中均匀分布，以减少冲突的发生。在本次实验中，我简单地将字符串中每个字符的ASCII值相加并取余作为哈希值，这种简单的哈希函数可能无法很好地处理特定数据集中的冲突。在实际应用中，需要根据具体情况选择更优的哈希函数，以提高哈希表的性能和效率。 | | |
| 实验小结 | 在本实验中，我通过实现两个类HashTable1和HashTable2，分别采用链地址法和线性探查法解决哈希表中的冲突问题，来加深对哈希表的理解，并验证其在实际应用中的效果。 | | |
| 数据结构  （自定义数据类型） | // 哈希表节点  struct HashNode {  long number;  string name;  string address;  HashNode \*next;  };  // 哈希表1：以姓名为关键字，采用链地址法解决冲突  class HashTable1 {  private:  static const int TABLE\_SIZE = 100; // 哈希表大小  HashNode \*table[TABLE\_SIZE]; // 哈希表数组  // 哈希函数：根据用户名计算哈希值  int hashFunction(string userName);  // 插入记录到哈希表  void insertRecord(long number, string name, string address);  // 根据用户名查找记录  HashNode \*searchName(string name);  };  // 哈希表2：以电话号码为关键字，采用线性探查法解决冲突  class HashTable2 {  private:  static const int TABLE\_SIZE = 100; // 哈希表大小  HashNode \*table[TABLE\_SIZE]; // 哈希表数组  // 哈希函数：根据电话号码计算哈希值  int hashFunction(long number);  // 插入记录到哈希表  void insertRecord(long number, string name, string address);  // 根据电话号码查找记录  HashNode \*searchNumber(long number);  }; | | |
| 主要算法  （或算法说明） | 链地址法：通过哈希函数将关键字映射到哈希表的索引位置，每个索引位置存储一个链表，链表中的节点包含相同哈希值的记录。插入操作将新记录插入到链表的头部，查找操作在对应的链表中查找目标记录。  线性探查法：通过哈希函数将关键字映射到哈希表的索引位置，若该位置已被占用，则线性探查下一个位置，直到找到空闲位置或找遍整个哈希表。插入操作将新记录插入到第一个空闲位置，查找操作从目标位置开始线性探查，直到找到目标记录或遇到空闲位置。 | | |