

**Lab Report**

**实验报告**

|  |  |
| --- | --- |
| **Course**: | Class Libraries and Data Structures |
| **Semester**: | 1st semester of the academic year **2024-2025** |
| **Major**: | Software Engineering |
| **Class**: | 2023 |
| **Student Name**: | 吴仲雄 |
| **Student ID:** | 222023321062007 |
| **Teacher:** | ZHAO, Hengjun (赵恒军) |

**School of Computer and Information Science**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Name | | Binary Search Tree  二分查找树 | | | |
| Date | | Nov，2024 | Type | | ☑Confirmatory （验证确认型）  ☑Design（设计型）  🗆Comprehensive（综合型） |
| 1. **Objective & Requirements（实验目的）**    1. Understand the concept and property of binary search tree   理解二分查找树的概念和性质   * 1. Get familiar with the insert, delete and find operations on binary search tree   熟悉二分查找树的节点插入、节点删除和元素查找操作   * 1. Grasp the design of recursive or iterative algorithms about binary search tree   掌握二分查找树上的递归和迭代算法的设计方法   * 1. Understand the concept of height of a binary search tree; can use iterative algorithms to compute the height of a binary search tree   掌握二分查找树的树高的定义，能利用递归或迭代算法计算二分查找树的树高   * 1. Understand the relationship between the size and height of a binary search tree; can deduce the relationship both mathematically and by experiment.   理解二分查找树树高和节点数目之间的关系，能够借助实验利用统计和方法定量分析二者之间的函数关系；能够对二者之间的关系进行数学上的推导。 | | | | | |
| 1. **Experimental environment (**platform and software**)（实验环境）**   Windows 7 (or higher versions) + Visual Studio 2010 (or higher versions) | | | | | |
| 1. Experimental content and design (Main Content, Procedure, Codes and Results)（此部分应包含每一个实验内容的详细设计，含实验思路、详细实验步骤、核心代码说明等）   Task 1   1. Complete the implementation of the BST container based on the code sent to you. In particular, the method you are required to implement includes:  * BinSearchTree(); // default constructor * int size() const; // get the number of stored elements * int height() const; // compute the height * Iterator insert(const T& item); //insertion * ~BinSearchTree(); //destructor   It is suggested that some of the methods be implemented in a recursive way.  请基于所给代码实现BST类，至少应实现如下方法：   * BinSearchTree(); // 默认构造函数 * int size() const; // 获取存储元素的数目 * int height() const; // 获取树高 * Iterator insert(const T& item); // 元素的插入 * ~BinSearchTree(); // 析构函数   其中的部分方法建议用递归算法实现。   1. Generate a series of integers randomly and insert them into an empty binary search tree, and compute the height of the tree. Repeat this for a number of times and compute the average height of a BST of size n. Try to discover the mathematical relationship between the size and the height of a BST based on your analysis.   生成一系列随机数，通过其从空树开始通过逐个插入的方式构造一个BST，计算该BST的树高；变换随机数序列的数值范围和数目，多次重复实验，获取n个节点BST平均树高的统计量；对所得结果进行数据拟合等分析，获取BST平均树高和节点数目之间的数学关系。  实验思路：  本实验旨在深入理解二叉搜索树（BST）的结构特性，特别是其高度与节点数量之间的关系。通过构建多个随机生成整数序列并插入到空的二叉搜索树中，计算每棵树的高度，并最终求得平均高度，以此来探索数学上的理论值与实验结果之间的联系。  详细实验步骤：   * 准备阶段：用户首先需要输入一系列参数，包括要插入元素的数量 n、随机数的最小值和最大值以及实验次数。 * 生成随机数列：基于用户提供的范围，使用随机数生成器创建一个长度为 n 的整数向量 randomNumbers。 * 构建二叉搜索树：对于每次实验，初始化一个新的 BinSearchTree<int> 对象，并将上述生成的随机数依次插入该树中。 * 记录树高：在所有数字都被成功插入后，调用 height() 方法获取当前树的高度，并将其累加至总高度变量 totalHeight 中。 * 重复实验：按照用户指定的实验次数重复以上过程。 * 计算平均高度：所有实验完成后，根据累积的高度值计算出平均树高，并输出这一统计结果。   以下是实现上述功能的核心代码(bst.h)：  bst.h:  #ifndef BST\_H  #define BST\_H  #include <iostream>  #include <string>  #define NULL 0  template<typename T>  class BinSearchTree  {  private:  struct Node  {  T item;  Node\* parent;  Node\* left;  Node\* right;  }; // Node  Node \*root;  int count;  public:  class Iterator; //very important!!!  // Postcondition: this BinSearchTree is empty.  BinSearchTree(); // default constructor  // Postcondition: the number of items in this BinSearchTree  // has been returned.  int size() const;  // Postcondition: the height of this BinSearchTree  // has been returned.  int height() const;  int heightHelper(Node\* node) const;  // Postcondition: item has been inserted into this BinSearchTree. An iterator  // positioned at the inserted item has been returned. The  // averageTime(n) is O(log n) and worstTime(n) is O(n).  Iterator insert(const T& item);  // Postcondition: if there is an item in this BinSearchTree that equals item,  // the value returned is an iterator pointing to that item.  // Otherwise, the value returned is an iterator with the same  // value as the iterator returned by the end( ) method. The  // averageTime(n) is O(log n) and worstTime(n) is O(n).  Iterator find(const T& item) const;  // Precondition: itr is positioned at an item in this BinSearchTree.  // Postcondition: the item that itr is positioned at has been deleted from  // this BinSearchTree. The averageTime(n) is O(log n)  // and worstTime(n) is O(n).  void erase(Iterator itr);  // Postcondition: The space allocated for this BinSearchTree has been  // deallocated. The worstTime(n) is O(n).  ~BinSearchTree();  void deleteTreeHelper(Node\* node);  // Postcondition: The tree-shape of this BinSearchTree has been printed  void printTree();  class Iterator  {  friend class BinSearchTree<T>;  private:  Node \*curr;  Iterator(Node \*currNode);  public:  Iterator();  Iterator& operator++ ();  Iterator& operator-- ();  T& operator\* () const;  bool operator== (const Iterator &otherIterator)const;  }; // Iterator  // Postcondition: if this BinSearchTree is non-empty, an iterator positioned  // at the smallest item in the tree has been returned.  // Otherwise, the iterator returned has the same value as the  // iterator returned by the end( ) method.  Iterator begin();  // Postcondition: the value returned is an iterator that can be used in a  // comparison for ending the traversal of this BinSearchTree.  // If this BinSearchTree is non-empty, the largest item is in the  // position just before the position of the iterator returned.  Iterator end();  }; // BinSearchTree  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*the following is the implmentation of the bst class interfaces\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  template<typename T>  BinSearchTree<T>::BinSearchTree()  {  root = nullptr;  count = 0;  }  template<typename T>  int BinSearchTree<T>::size() const  {  return count;  }  template<typename T>  int BinSearchTree<T>::height() const  {  return heightHelper(root);  }  template<typename T>  int BinSearchTree<T>::heightHelper(Node\* node) const  {  if (!node) return -1;  return 1 + std::max(heightHelper(node->left), heightHelper(node->right));  }  template<typename T>  typename BinSearchTree<T>::Iterator BinSearchTree<T>::insert(const T& item)  {  Node\* newNode = new Node{ item, nullptr, nullptr, nullptr };  if (!root) {  root = newNode;  count++;  return Iterator(newNode);  }  Node\* current = root;  Node\* parent = nullptr;  while (current) {  parent = current;  if (item < current->item) {  current = current->left;  }  else {  current = current->right;  }  }  if (item < parent->item) {  parent->left = newNode;  }  else {  parent->right = newNode;  }  newNode->parent = parent;  count++;  return Iterator(newNode);  }  template<typename T>  void BinSearchTree<T>::printTree()  {  //Please think about how to implement this.  }  template<typename T>  typename BinSearchTree<T>::Iterator BinSearchTree<T>::find(const T& item) const  {  //not finished  return Iterator();  }  template<typename T>  void BinSearchTree<T>::erase(Iterator itr)  {  //not finished  }  template<typename T>  BinSearchTree<T>::~BinSearchTree()  {  deleteTreeHelper(root);  }  template<typename T>  void BinSearchTree<T>::deleteTreeHelper(Node\* node)  {  if (node == nullptr) { // if the tree is empty, return  return;  }  deleteTreeHelper(node->left);  deleteTreeHelper(node->right);  delete node;  }  template<typename T>  typename BinSearchTree<T>::Iterator BinSearchTree<T>::begin()  {  //not finished  return Iterator();  }  template<typename T>  typename BinSearchTree<T>::Iterator BinSearchTree<T>::end()  {  //not finished  return Iterator();  }  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*the following is the implmentation of the iterator inner class\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  template<typename T>  BinSearchTree<T>::Iterator::Iterator(Node \*currNode)  {  curr = currNode;  }  template<typename T>  BinSearchTree<T>::Iterator::Iterator( )  {  }  template<typename T>  typename BinSearchTree<T>::Iterator& BinSearchTree<T>::Iterator::operator++()  {  //not finished  return Iterator();  }  template<typename T>  typename BinSearchTree<T>::Iterator& BinSearchTree<T>::Iterator::operator--()  {  //not finished  return Iterator();  }  template<typename T>  T& BinSearchTree<T>::Iterator::operator\* () const  {  return curr->item;  }  template<typename T>  bool BinSearchTree<T>::Iterator::operator==(const Iterator &otherIterator) const  {  //not finished  return false;  }  #endif  main.cpp:  #include <iostream>  #include <vector>  #include <random>  #include <ctime>  #include "bst.h"  using namespace std;  vector<int> generateRandomNumbers(int size, int minVal, int maxVal) {  random\_device rd;  mt19937 gen(rd());  uniform\_int\_distribution<> dis(minVal, maxVal);  vector<int> numbers;  for (int i = 0; i < size; ++i) {  numbers.push\_back(dis(gen));  }  return numbers;  }  int main() {  int n, minVal, maxVal, numExperiments;  cout << "Enter the number of elements to insert (n): ";  cin >> n;    cout << "Enter the minimum value for random numbers: ";  cin >> minVal;  cout << "Enter the maximum value for random numbers: ";  cin >> maxVal;  cout << "Enter the number of experiments: ";  cin >> numExperiments;  double totalHeight = 0; // used to calculate the average height  for (int i = 0; i < numExperiments; ++i) {  BinSearchTree<int> bst; // create a new binary search tree  vector<int> randomNumbers = generateRandomNumbers(n, minVal, maxVal);  cout << "Experiment " << i + 1 << ": Inserting numbers: ";  for (int num : randomNumbers) {  cout << num << " ";  bst.insert(num); // insert the random number into the tree  }  cout << endl;  int height = bst.height(); // get the height of the tree  cout << "Tree height after insertion: " << height << endl;  totalHeight += height; // add the height to the total height  }  double averageHeight = totalHeight / numExperiments;  cout << "Average tree height after " << numExperiments << " experiments: " << averageHeight << endl;  return 0;  } | | | | | |
| 1. **Result analysis and discussion**（Analysis of experimental results and summing up the harvest and the existing problems）（此部分应包含实验结果，对实验结果的分析，实验收获的总结，实验中存在问题的讨论等；另外，需要回应一下如下思考题：   1）如何推导证明Binary Search Tree的平均树高与结点数成对数关系？  2）如何实现树状打印方法  void printTree(); // print the tree-shape of the BST  以获取类似    的结果？  ）  实验结果：  进行实验，固定随机数范围为1-99，节点数目按5、10、20、40、60、80、100、200、300、400、500递增，每次实验1000次。结果如下：  5个节点：    10个节点：    20个节点：    40个节点：    60个节点：    80个节点：    100个节点：    200个节点：    300个节点：    400个节点：    500个节点：    对所得结果进行数据拟合分析：  查看图片  查看图片  能看出对数模型的拟合效果较好。  得出BST平均树高(y)和节点数目(x)间数学关系为：y=−4.432+3.812ln(x)  结果分析：  通过实验，我们验证了二分查找树的平均树高与节点数目之间呈现对数关系。随着节点数目的增加，平均树高呈现缓慢增长趋势，符合对数函数的特性。数据拟合结果显示，对数模型对实验数据的拟合效果较好，说明对数关系是描述 BST 平均树高与节点数目之间关系的合理模型。实验结果表明，BST 的平均树高与节点数目之间并非线性关系，而是随着节点数目的增加，增长速度逐渐减慢。  收获总结：  在本次实验中我深入理解了二分查找树的概念、性质和操作方法，包括插入、删除和查找等操作。掌握了二分查找树递归和迭代算法的设计方法，并能够运用递归算法实现 BST 的相关操作。理解了 BST 树高的概念，并能够利用递归算法计算 BST 的树高。通过实验和数据分析，验证了 BST 平均树高与节点数目之间的对数关系，并掌握了利用数据拟合方法分析实验数据的能力。  问题讨论：   * 实验中生成的随机数序列可能存在一定的规律性，对实验结果产生一定的影响。为了更准确地验证 BST 平均树高与节点数目之间的关系，可以采用不同的随机数生成方法，或者增加实验次数。 * 实验中只考虑了 BST 的平均树高，而没有考虑 BST 的最坏情况树高。在实际应用中，需要考虑 BST 的最坏情况性能，并进行相应的优化。 * 实验中使用的 BST 实现只包含了插入操作，没有包含删除和查找操作。在实际应用中，需要实现 BST 的完整功能。   思考题：  (1) 可以通过数学归纳法证明 BST 的平均树高与节点数目成对数关系。首先，对于只有根节点的 BST，树高为 0，即 log(1) = 0。假设对于节点数目为 n 的 BST，平均树高为 log(n)。那么，对于节点数目为 n+1 的 BST，可以将它看作是在节点数目为 n 的 BST 上插入一个节点。插入操作需要比较节点值，并将新节点插入到合适的位置。在最坏情况下，新节点需要插入到 BST 的最深处，即树高增加 1。因此，节点数目为 n+1 的 BST 的平均树高为 log(n) + 1 = log(n+1)。由此可以证明，对于任意节点数目的 BST，平均树高与节点数目成对数关系。  (2) 可以使用递归算法实现树状打印方法。具体实现方法如下：   * 对于空树，直接return。 * 对于非空树，首先打印左子树，然后打印当前节点的值，最后打印右子树。 * 在打印左子树和右子树时，需要在每个节点前面添加适当的空格或缩进，以表示节点的层级关系。   可以使用以下代码实现树状打印方法：  template<typename T>  void BinSearchTree<T>::printTree() {  printTreeHelper(root, 0);  }  template<typename T>  void BinSearchTree<T>::printTreeHelper(Node\* node, int level) {  if (node == nullptr) {  return;  }  // 打印右子树  printTreeHelper(node->right, level + 1);  // 打印当前节点  for (int i = 0; i < level; ++i) {  std::cout << " "; // 每一层增加两个空格的缩进  }  std::cout << node->item << std::endl;  // 打印左子树  printTreeHelper(node->left, level + 1);  }  这个实现中，printTreeHelper 函数接受两个参数：当前节点 node 和当前的层级 level。层级用于确定打印节点时需要的缩进量。函数首先递归地打印右子树，然后打印当前节点（在适当的缩进后），最后递归地打印左子树。 | | | | | |
| Comments & Evaluation | Content & Design (A-E) | | |  | |
| Procedure & Codes (A-E) | | |  | |
| Results (A-E) | | |  | |
| Analysis & Discussion (A-E) | | |  | |
| Score (A-E):  Feedback comments: | | | | |