2022113573 张宇杰

1. 设计 BST 的左右链存储结构，并实现 BST 插入（建立）、删除、查找和排序算法

插入：

输入程序：

1. #include <iostream>
2. #include ".\src\BinarySearch"
3. #include ".\src\BinarySearchTree"
4. int main()
5. {
6. system("chcp 65001"); *// set terminal to UTF-8*
7. int data[] = { 5, 6, 2, 15, 64, 32, 18, 60, 94, -2 };
8. BinaryTree::BST<int> tree;
9. for (int i = 0; i < 10; ++i)
10. tree.insert(data[i]);
11. tree.show();
12. return 0;
13. }

输出：

文本

描述已自动生成

删除：

输入程序：

1. #include <iostream>
2. #include ".\src\BinarySearch"
3. #include ".\src\BinarySearchTree"
4. int main()
5. {
6. system("chcp 65001"); *// set terminal to UTF-8*
7. int data[] = { 5, 6, 2, 15, 64, 32, 18, 60, 94, -2, 12, 100, 89, 4 };
8. BinaryTree::BST<int> tree;
9. for (int i = 0; i < 14; ++i)
10. tree.insert(data[i]);
11. tree.show();
12. tree.remove(64);
13. tree.show();
14. return 0;
15. }

输出：

图示, 示意图

描述已自动生成

查找：

输入程序：

1. #include <iostream>
2. #include ".\src\BinarySearch"
3. #include ".\src\BinarySearchTree"
4. int main()
5. {
6. system("chcp 65001"); *// set terminal to UTF-8*
7. int data[] = { 5, 6, 2, 15, 64, 32, 18, 60, 94, -2, 12, 100, 89, 4 };
8. BinaryTree::BST<int> tree;
9. for (int i = 0; i < 14; ++i)
10. tree.insert(data[i]);
11. tree.show();
12. std::cout << tree.contain(32) << std::endl;
13. std::cout << tree.contain(40) << std::endl;
14. return 0;
15. }

输出：

图示, 示意图

描述已自动生成

排序：

输入程序：

1. #include <iostream>
2. #include ".\src\BinarySearch"
3. #include ".\src\BinarySearchTree"
4. int main()
5. {
6. system("chcp 65001"); *// set terminal to UTF-8*
7. int data[] = { 5, 6, 2, 15, 64, 32, 18, 60, 94, -2, 12, 100, 89, 4 };
8. BinaryTree::BST<int> tree;
9. for (int i = 0; i < 14; ++i)
10. tree.insert(data[i]);
11. tree.show();
12. tree.sort(std::cout);
13. return 0;
14. }

输出：

图片包含 文本

描述已自动生成

1. 实现折半查找的递归和非递归算法：

非递归：

输入程序：

1. #include <iostream>
2. #include ".\src\BinarySearch"
3. #include ".\src\BinarySearchTree"
4. int main()
5. {
6. system("chcp 65001"); *// set terminal to UTF-8*
7. int arr[] = { 1, 1, 2, 3, 4, 4, 5, 5, 7, 7, 7, 8, 8, 9, 9, 9, 9, 10 };
8. int\* first = ::My::lower\_bound(arr, arr + 18, 7);
9. int\* last = ::My::upper\_bound(arr, arr + 18, 7);
10. std::cout << "7: [" << first - arr << ", " << last - arr << ')' << std::endl;
11. first = ::My::lower\_bound(arr, arr + 18, 9);
12. last = ::My::upper\_bound(arr, arr + 18, 9);
13. std::cout << "9: [" << first - arr << ", " << last - arr << ')' << std::endl;
14. return 0;
15. }

输出：

文本

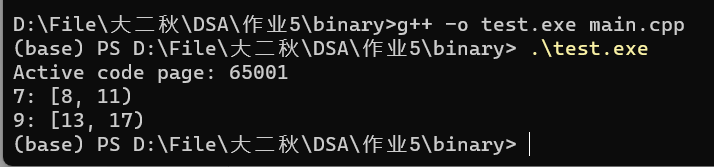
描述已自动生成

递归：

输入程序：

1. #include <iostream>
2. #include ".\src\BinarySearch"
3. #include ".\src\BinarySearchTree"
4. int main()
5. {
6. system("chcp 65001"); *// set terminal to UTF-8*
7. int arr[] = { 1, 1, 2, 3, 4, 4, 5, 5, 7, 7, 7, 8, 8, 9, 9, 9, 9, 10 };
8. int\* first = ::My::lower\_bound\_rec(arr, arr + 18, 7, std::less<int>());
9. int\* last = ::My::upper\_bound\_rec(arr, arr + 18, 7, std::less<int>());
10. std::cout << "7: [" << first - arr << ", " << last - arr << ')' << std::endl;
11. first = ::My::lower\_bound\_rec(arr, arr + 18, 9, std::less<int>());
12. last = ::My::upper\_bound\_rec(arr, arr + 18, 9, std::less<int>());
13. std::cout << "9: [" << first - arr << ", " << last - arr << ')' << std::endl;
14. return 0;
15. }

输出：

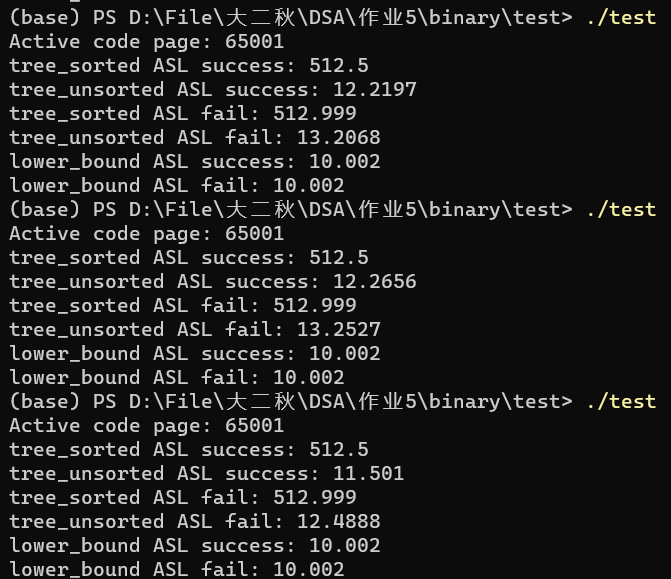


1. 实验比较：设计并产生实验测试数据，考察比较两种查找方法的时间性能，并与理论结果进行比较

输入程序：可见于./binary/test/main.cpp

1. #include <iostream>
2. #include <algorithm>
3. #include <time.h>
4. #include "..\src\BinarySearch"
5. #include "..\src\BinarySearchTree"
6. template <typename Val, class ForwardIter>
7. int lower\_bound\_search\_len(ForwardIter first, ForwardIter last, Val key) {
8. int ret = 0;
9. while (first < last) {
10. ForwardIter mid = first + (last - first) / 2;
11. if (++ret, \*mid < key)
12. first = mid + 1;
13. else
14. last = mid;
15. }
16. return ret;
17. }
18. int main()
19. {
20. system("chcp 65001"); *// set terminal to UTF-8*
22. int data\_sorted[1024], data\_unsorted[1024];
23. for (int i = 0; i < 1024; ++i)
24. data\_unsorted[i] = data\_sorted[i] = 2 \* i + 1;
25. srand(time(0));
26. std::random\_shuffle(data\_unsorted, data\_unsorted + 1024);
27. int data\_fail[1025];
28. for (int i = 0; i < 1025; ++i)
29. data\_fail[i] = 2 \* i;
30. BinaryTree::BST<int> tree\_sorted, tree\_unsorted;
31. for (int i = 0; i < 1024; ++i) {
32. tree\_sorted.insert(data\_sorted[i]);
33. tree\_unsorted.insert(data\_unsorted[i]);
34. }
35. int sum1 = 0, sum2 = 0;
36. for (int i = 0; i < 1024; ++i) {
37. sum1 += tree\_sorted.search\_len(data\_sorted[i]);
38. sum2 += tree\_unsorted.search\_len(data\_unsorted[i]);
39. }
40. std::cout << "tree\_sorted ASL success: " << sum1 / 1024.0 << std::endl;
41. std::cout << "tree\_unsorted ASL success: " << sum2 / 1024.0 << std::endl;
42. sum1 = sum2 = 0;
43. for (int i = 0; i < 1025; ++i) {
44. sum1 += tree\_sorted.search\_len(data\_fail[i]);
45. sum2 += tree\_unsorted.search\_len(data\_fail[i]);
46. }
47. std::cout << "tree\_sorted ASL fail: " << sum1 / 1025.0 << std::endl;
48. std::cout << "tree\_unsorted ASL fail: " << sum2 / 1025.0 << std::endl;
49. sum1 = 0;
50. for (int i = 0; i < 1024; ++i)
51. sum1 += lower\_bound\_search\_len(data\_sorted, data\_sorted + 1024, data\_sorted[i]);
52. std::cout << "lower\_bound ASL success: " << sum1 / 1024.0 << std::endl;
53. sum2 = 0;
54. for (int i = 0; i < 1025; ++i)
55. sum2 += lower\_bound\_search\_len(data\_sorted, data\_sorted + 1024, data\_fail[i]);
56. std::cout << "lower\_bound ASL fail: " << sum2 / 1025.0 << std::endl;
57. return 0;
58. }

输出：（三次）



分析：

sorted\_BST:

理论上失败ASL为1+n/2-1/(n+1)=512.9990244，实际测试为512.999

理论上成功ASL为(n+1)/2=512.5，实际测试为512.5

unsorted\_BST:

理论上失败ASL（满二叉树）约为log2(n+1)=10.0014，实际测试约为12.9828

理论上成功ASL（满二叉树）约为log2(n+1)-1=9.0014，实际测试约为11.9954

折半：

理论上失败ASL（满二叉判定树）约为log2(n+1)=10.0014，实际测试约为10.002

理论上成功ASL（满二叉判定树）约为log2(n+1)-1=9.0014，实际测试约为10.002

（可能是因为实现的是lower\_bound而不是naïve的binary\_search造成的）

（5）以上实验能否说明：就平均性能而言，BST 查找与折半查找差不多，为什么

可以认为。因为BST查找和折半查找的平均时间复杂度都是O(logn),但由于我们不能保证BST是一个满二叉树，导致在实际情况下BST的性能比折半要差上一点。我认为，当数据量足够大时，这种差异是常数级的

作业题目 2 ：简答题 （选做）

按题目要求回答下列问题：

1. 比较说明堆和二叉排序树的区别。

堆要求父节点均比两个子节点大（小），而二叉排序树要求父节点大于（小于）左子树的使用节点，而小于（大于）右子树的使用节点

1. 若只想得到一个序列中第 k (k ≥ 5) 个最小元素之前的部分排序序列，则最好采用什么排序方法?

堆排序。参考STL中partial\_sort源码

1. template <class RandomAccessIterator>
2. inline void partial\_sort(RandomAccessIterator first, RandomAccessIterator middle, RandomAccessIterator last) {
3. \_\_partial\_sort(first, middle, last, value\_type(first));
4. }
6. template <class RandomAccessIterator, class T>
7. void \_\_partial\_sort(RandomAccessIterator first, RandomAccessIterator middle, RandomAccessIterator last, T\*) {
8. make\_heap(first, middle); *//将区间[first, middle)构造为一个堆结构*
9. for (RandomAccessIterator i = middle; i < last; ++i)
10. if (\*i < \*first)    *// 遍历堆以外的元素，并将更优的元素放入堆中*
11. \_\_pop\_heap(first, middle, i, T(\*i), distance\_type(first)); *// first值放i中，i的原值融入heap并调整*
12. sort\_heap(first, middle); *// 对最终的堆进行排序*
13. }

复杂度约为：(last-first)log2(middle-first)

作业题目 3

1. 给出算法的基本设计思想

显然，将数组排序后，前一半元素作为A1，剩下作为A2即可满足要求

但我们只需要获得A1，而不需要真正对整个数组进行排序，其复杂度为O(nlogn)

利用快速排序的思想，当使用pivot对数组进行一次遍历后，pivot之前的数都比pivot小，pivot之后的数都比pivot大

如果此时pivot的下标正好为floor(n/2),那么算法结束

否则像真正的快速排序一样递归的寻找pivot

这样是正确的

对于n为偶数，就是将原数组对半分

对于n为奇数，有(中位数之前)(中位数)(中位数之后)三部分，我们使之大致按照升序排序，由于算法结束条件为floor(n/2)，所以会把(中位数之前)作为A1,而(中位数)(中位数之后)作为A2，这样就使两集合的和之差最大

（2） 根据设计思想，采用 C/C++/Java 等程序语言描述算法，关键之处给出注释

懒了，不想写

1. 说明你所设计算法的平均时间复杂度和空间复杂度

时间复杂度应该是O(n)，空间复杂度为O(1)