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

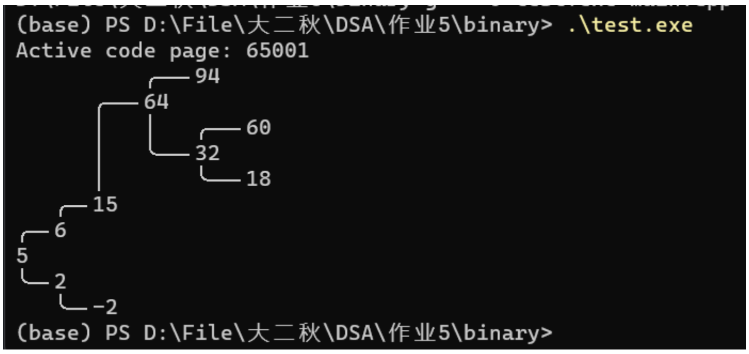
插入：

输入程序：

```cpp
#include <iostream>
#include ".\src\BinarySearch"
#include ".\src\BinarySearchTree"

int main()
{
    system("chcp 65001"); // set terminal to UTF-8
    int data[] = { 5, 6, 2, 15, 64, 32, 18, 60, 94, -2 };
    BinaryTree::BST<int> tree;
    for (int i = 0; i < 10; ++i)
        tree.insert(data[i]);
    tree.show();
    return 0;
}
```

输出：




删除：

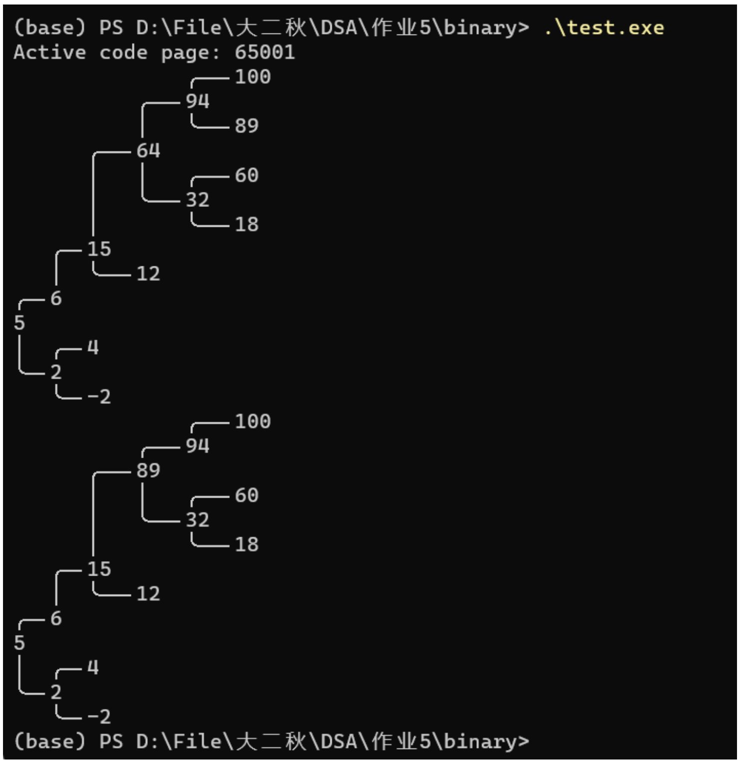
输入程序：

```cpp
#include <iostream>
#include ".\src\BinarySearch"
#include ".\src\BinarySearchTree"

int main()
{
    system("chcp 65001"); // set terminal to UTF-8
    int data[] = { 5, 6, 2, 15, 64, 32, 18, 60, 94, -2, 12, 100, 89, 4 };
    BinaryTree::BST<int> tree;
    for (int i = 0; i < 14; ++i)
```

```
        tree.insert(data[i]);
    tree.show();
    tree.remove(64);
    tree.show();
    return 0;
}
```

输出：



```
(base) PS D:\File\大二秋\DSA\作业5\binary> .\test.exe
Active code page: 65001
                ┌──100
            ┌──94
            │   └──89
        ┌──64
        │   │   ┌──60
        │   └──32
        │       └──18
    ┌──15
    │   └──12
┌──6
5
│   ┌──4
└──2
    └──-2
                ┌──100
            ┌──94
        ┌──89
        │   │   ┌──60
        │   └──32
        │       └──18
    ┌──15
    │   └──12
┌──6
5
│   ┌──4
└──2
    └──-2
(base) PS D:\File\大二秋\DSA\作业5\binary>
```

查找：

输入程序：

```
#include <iostream>
#include ".\src\BinarySearch"
#include ".\src\BinarySearchTree"

int main()
{
    system("chcp 65001"); // set terminal to UTF-8
    int data[] = { 5, 6, 2, 15, 64, 32, 18, 60, 94, -
      2, 12, 100, 89, 4 };
    BinaryTree::BST<int> tree;
    for (int i = 0; i < 14; ++i)
        tree.insert(data[i]);
    tree.show();
    std::cout << tree.contain(32) << std::endl;
    std::cout << tree.contain(40) << std::endl;
```

```cpp
        return 0;
    }
```

输出：

```
(base) PS D:\File\大二秋\DSA\作业5\binary> .\test.exe
Active code page: 65001
                ┌───100
            ┌───94
            │   └───89
        ┌───64
        │   │   ┌───60
        │   └───32
        │       └───18
    ┌───15
    │   └───12
┌───6
5
│   ┌───4
└───2
    └───-2
1
0
(base) PS D:\File\大二秋\DSA\作业5\binary>
```

排序：

输入程序：

```cpp
#include <iostream>
#include ".\src\BinarySearch"
#include ".\src\BinarySearchTree"

int main()
{
    system("chcp 65001"); // set terminal to UTF-8
    int data[] = { 5, 6, 2, 15, 64, 32, 18, 60, 94, -
      2, 12, 100, 89, 4 };
    BinaryTree::BST<int> tree;
    for (int i = 0; i < 14; ++i)
        tree.insert(data[i]);
    tree.show();
    tree.sort(std::cout);
    return 0;
}
```

输出：

```
(base) PS D:\File\大二秋\DSA\作业5\binary> .\test.exe
Active code page: 65001
                  ┌──100
              ┌──94
              │   └──89
          ┌──64
          │   │   ┌──60
          │   └──32
          │       └──18
      ┌──15
      │   └──12
  ┌──6
5
  │   ┌──4
  └──2
      └──-2
-2 2 4 5 6 12 15 18 32 60 64 89 94 100
```

2、实现折半查找的递归和非递归算法：

非递归：

输入程序：

```cpp
#include <iostream>
#include ".\src\BinarySearch"
#include ".\src\BinarySearchTree"

int main()
{
    system("chcp 65001"); // set terminal to UTF-8
    int arr[] = { 1, 1, 2, 3, 4, 4, 5, 5, 7, 7, 7, 8, 8, 9, 9, 9, 9, 10 };
    int* first = ::My::lower_bound(arr, arr + 18, 7);
    int* last = ::My::upper_bound(arr, arr + 18, 7);
    std::cout << "7: [" << first - arr << ", " << last - arr << ')' << std::endl;
    first = ::My::lower_bound(arr, arr + 18, 9);
    last = ::My::upper_bound(arr, arr + 18, 9);
    std::cout << "9: [" << first - arr << ", " << last - arr << ')' << std::endl;
    return 0;
}
```

输出：

```
(base) PS D:\File\大二秋\DSA\作业5\binary> .\test.exe
Active code page: 65001
7: [8, 11)
9: [13, 17)
(base) PS D:\File\大二秋\DSA\作业5\binary>
```

递归：

输入程序：

```cpp
#include <iostream>
```

```cpp
#include ".\src\BinarySearch"
#include ".\src\BinarySearchTree"

int main()
{
    system("chcp 65001"); // set terminal to UTF-8
    int arr[] = { 1, 1, 2, 3, 4, 4, 5, 5, 7, 7, 7, 8, 8, 9, 9, 9, 9, 10 };
    int* first = ::My::lower_bound_rec(arr, arr + 18, 7, std::less<int>());
    int* last = ::My::upper_bound_rec(arr, arr + 18, 7, std::less<int>());
    std::cout << "7: [" << first - arr << ", " << last - arr << ')' << std::endl;
    first = ::My::lower_bound_rec(arr, arr + 18, 9, std::less<int>());
    last = ::My::upper_bound_rec(arr, arr + 18, 9, std::less<int>());
    std::cout << "9: [" << first - arr << ", " << last - arr << ')' << std::endl;
    return 0;
}
```

输出：

```
D:\File\大二秋\DSA\作业5\binary>g++ -o test.exe main.cpp
(base) PS D:\File\大二秋\DSA\作业5\binary> .\test.exe
Active code page: 65001
7: [8, 11)
9: [13, 17)
(base) PS D:\File\大二秋\DSA\作业5\binary>
```

3、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

```cpp
#include <iostream>
#include <algorithm>
#include <time.h>
#include "..\src\BinarySearch"
#include "..\src\BinarySearchTree"

template <typename Val, class ForwardIter>
int lower_bound_search_len(ForwardIter first, ForwardIter last, Val key) {
    int ret = 0;
    while (first < last) {
```

```cpp
        ForwardIter mid = first + (last - first) / 2;
        if (++ret, *mid < key)
            first = mid + 1;
        else
            last = mid;
    }
    return ret;
}

int main()
{
    system("chcp 65001"); // set terminal to UTF-8

    int data_sorted[1024], data_unsorted[1024];
    for (int i = 0; i < 1024; ++i)
        data_unsorted[i] = data_sorted[i] = 2 * i + 1;
    srand(time(0));
    std::random_shuffle(data_unsorted, data_unsorted + 1024);

    int data_fail[1025];
    for (int i = 0; i < 1025; ++i)
        data_fail[i] = 2 * i;

    BinaryTree::BST<int> tree_sorted, tree_unsorted;
    for (int i = 0; i < 1024; ++i) {
        tree_sorted.insert(data_sorted[i]);
        tree_unsorted.insert(data_unsorted[i]);
    }

    int sum1 = 0, sum2 = 0;
    for (int i = 0; i < 1024; ++i) {
        sum1 += tree_sorted.search_len(data_sorted[i]);
        sum2 += tree_unsorted.search_len(data_unsorted[i]);
    }
    std::cout << "tree_sorted ASL success: " << sum1 / 1024.0 << std::endl;
    std::cout << "tree_unsorted ASL success: " << sum2 / 1024.0 << std::endl;

    sum1 = sum2 = 0;
    for (int i = 0; i < 1025; ++i) {
        sum1 += tree_sorted.search_len(data_fail[i]);
        sum2 += tree_unsorted.search_len(data_fail[i]);
    }
```

```cpp
    std::cout << "tree_sorted ASL fail: " << sum1 / 1025.0 << std::endl;
    std::cout << "tree_unsorted ASL fail: " << sum2 / 1025.0 << std::endl;

    sum1 = 0;
    for (int i = 0; i < 1024; ++i)
        sum1 += lower_bound_search_len(data_sorted, data_sorted + 1024, data_sorted[i]);
    std::cout << "lower_bound ASL success: " << sum1 / 1024.0 << std::endl;

    sum2 = 0;
    for (int i = 0; i < 1025; ++i)
        sum2 += lower_bound_search_len(data_sorted, data_sorted + 1024, data_fail[i]);
    std::cout << "lower_bound ASL fail: " << sum2 / 1025.0 << std::endl;
    return 0;
}
```

输出：（三次）

```
(base) PS D:\File\大二秋\DSA\作业5\binary\test> ./test
Active code page: 65001
tree_sorted ASL success: 512.5
tree_unsorted ASL success: 12.2197
tree_sorted ASL fail: 512.999
tree_unsorted ASL fail: 13.2068
lower_bound ASL success: 10.002
lower_bound ASL fail: 10.002
(base) PS D:\File\大二秋\DSA\作业5\binary\test> ./test
Active code page: 65001
tree_sorted ASL success: 512.5
tree_unsorted ASL success: 12.2656
tree_sorted ASL fail: 512.999
tree_unsorted ASL fail: 13.2527
lower_bound ASL success: 10.002
lower_bound ASL fail: 10.002
(base) PS D:\File\大二秋\DSA\作业5\binary\test> ./test
Active code page: 65001
tree_sorted ASL success: 512.5
tree_unsorted ASL success: 11.501
tree_sorted ASL fail: 512.999
tree_unsorted ASL fail: 12.4888
lower_bound ASL success: 10.002
lower_bound ASL fail: 10.002
```

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
2. 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

```
template <class RandomAccessIterator>
inline void partial_sort(RandomAccessIterator first, RandomAccessIterator middle, RandomAccessIterator last) {
    __partial_sort(first, middle, last, value_type(first));
}

template <class RandomAccessIterator, class T>
void __partial_sort(RandomAccessIterator first, RandomAccessIterator middle, RandomAccessIterator last, T*) {
    make_heap(first, middle); //将区间[first, middle)构造为一个堆结构
        for (RandomAccessIterator i = middle; i < last; ++i)
            if (*i < *first)    // 遍历堆以外的元素，并将更优的元素放入堆中
    __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
    sort_heap(first, middle); // 对最终的堆进行排序
}
```

   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

（2） 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)