1 堆排序核心代码

我们分三步来实现堆排序算法。

1.1 sift_down 函数

```
template <typename T>
void sift_down(std::vector<T> &heap, int start, int end) {
    int parent = start;         // 父节点下标
    int child = 2 * parent + 1; // 左孩子下标

    while (child <= end) { // 防止越界
        if (child + 1 <= end && heap[child] < heap[child + 1]) {
            child += 1; // 选择较大的孩子
        }

        if (heap[parent] < heap[child]) {
            std::swap(heap[parent], heap[child]); // 调整父子节点
            parent = child;
            child = 2 * parent + 1;
        } else {
            break;
        }
    }
}
```

我们先初始化父子节点，要交换子节点为父节点下标的两倍加一或两倍加二，这由两个子节点的比较得出。(前提是没有越界）然后我们不断比较父节点和子节点的大小，如果父节点小于子节点，我们就交换两者的值，否则堆性质满足，退出循环。

1.2 heapify 函数

```
template <typename T>
void heapify(std::vector<T> &heap) {
    int n = heap.size();
    for (int i = n / 2 - 1; i >= 0; --i) {
        sift_down(heap, i, n - 1);
    } // 从最后一个子节点的父节点开始调整
}
```

这里 for 循环的起始位置是 $\lfloor \frac{n}{2} \rfloor$，意思是从最后一个子节点的父节点开始，逐渐向前遍历，完成堆的构建。

1.3 heap_sort 函数

```
template <typename T>
void heap_sort(std::vector<T> &arr) {
```

```
    Timer timer; // 开始计时
    heapify(arr); // 建堆

    for (int i = arr.size() - 1; i > 0; --i) {
        std::swap(arr[0], arr[i]); // 将堆顶元素与最后一个元素交换
        sift_down(arr, 0, i - 1);  // 调整堆
    }
    // 计时器在析构函数中自动停止并输出时间
}
```

接下来会介绍辅助函数，包括计时器。开始计时后，进行堆的构建，然后不断交换堆顶元素（即当前堆的最大元素）和最后一个元素，然后调整堆，直到堆的大小为 1，进行了原地的升序排序。完成后计时器会自动停止并输出时间。

2 辅助方法

2.1 计时器类

```
class Timer {
    public:
        Timer() : start_time_point(std::chrono::high_resolution_clock::now()) {}

        ~Timer() {
            stop();
        }

        void stop() {
            auto end_time_point = std::chrono::high_resolution_clock::now();
            auto start = std::chrono::time_point_cast<std::chrono::microseconds>(
                start_time_point).time_since_epoch().count();
            auto end = std::chrono::time_point_cast<std::chrono::microseconds>(end_time_point).
                time_since_epoch().count();

            auto duration = end - start;
            double seconds = duration * 0.000001;

            std::cout << std::fixed << std::setprecision(4) << "Duration: " << seconds << "
                seconds" << std::endl;
        }

    private:
        std::chrono::time_point<std::chrono::high_resolution_clock> start_time_point;
    };
```

我们实现的计时器在析构函数中会自动停止并输出时间，我们只需要在排序函数前创建一个计时器对象即可。

函数执行完毕后，计时器对象会自动析构，输出时间。

方法的关键是使用两个时间点构建 `start` 和 `end` 对象，相减后，转换为秒数输出，结果为秒数保留四位小数。

2.2 验证排序正确性

```
template <typename T>
bool check_sort(const std::vector<T> &arr) {
    for (size_t i = 1; i < arr.size(); ++i) {
        if (arr[i - 1] > arr[i]) {
            return false;
        }
    }
    return true;
}
```

我们的程序是升序排序，这里只需检查相邻元素是否递增即可，如果有逆序对，返回 `false`，否则返回 `true`。

2.3 与标准堆排序比较

```
void stl_heap_sort(std::vector<T> &arr) {
    Timer timer; // 开始计时
    std::make_heap(arr.begin(), arr.end());
    std::sort_heap(arr.begin(), arr.end());
    // 计时器在析构函数中自动停止并输出时间
}

template <typename T>
void compare(std::vector<T> &arr) {
    std::vector<T> arr1(arr);
    std::vector<T> arr2(arr);

    std::cout << "My Heap Sort: ";
    heap_sort(arr1);

    std::cout << "STL Heap Sort: ";
    stl_heap_sort(arr2);

    if (check_sort(arr1) && check_sort(arr2)) {
        std::cout << "Two sorting algorithms are correct." << std::endl;
    } else {
        std::cout << "Two sorting algorithms are wrong." << std::endl;
    }
    std::cout << std::endl;
}
```

首先使用一个函数封装有计时器的 STL 堆排序，然后我们比较两种排序算法的结果，最后我们检验了排序的正确性。

2.4 测试辅助函数

```
// 生成测试序列
enum class Order {
    ASC, // 升序
    DESC, // 降序
    RANDOM // 随机
};

enum class Repeat {
    NO, // 无重复
    YES // 有重复
};
```

这里是测试序列的类型，我们对整数和浮点数都进行了六次测试，使用枚举变量排列组合。

```
template <typename T>
void fill_data(std::vector<T>& arr, std::function<T()> generator, Repeat repeat) {
    if (repeat == Repeat::NO) {
        std::unordered_set<T> unique_elements;
        for (long long i = 0; i < arr.size(); ++i) {
            T value;
            do {
                value = generator();
            } while (unique_elements.find(value) != unique_elements.end());
            unique_elements.insert(value);
            arr[i] = value;
        }
    } else {
        for (long long i = 0; i < arr.size(); ++i) {
            arr[i] = generator();
        }
    }
}
```

这段代码我们用来填充数据，`generator` 是一个返回随机数的函数，用于生成对应类型的序列。稍后我们会看到，我们使用了 `std::uniform_int_distribution` 和 `std::uniform_real_distribution` 来生成随机数。在 `Repeat::NO` 的情况下，我们使用 `std::unordered_set` 来保证没有重复元素。否则，我们直接生成随机数，插入到参数 `arr` 中。

下面是生成整数的方法:

```
std::vector<int> generate_data_int(long long size, Order order, Repeat repeat) {
    std::vector<int> arr(size);
    std::random_device rd;
```

```
    std::mt19937 gen(rd());
    std::uniform_int_distribution<int> dis(1, size);

    auto generator = [&]() { return dis(gen); };
    fill_data<int>(arr, generator, repeat);

    if (order == Order::ASC) {
        std::sort(arr.begin(), arr.end());
    } else if (order == Order::DESC) {
        std::sort(arr.begin(), arr.end(), std::greater<int>());
    }

    return arr;
}
```

在 C++ 中生成随机数，我们先随机数种子，比如这里的 `std::random_device`，然后用 `std::mt19937` 引擎生成随机数，最后用 `std::uniform_int_distribution` 标准正态分布生成随机数。

填充数据方法的参数需要一个函数对象，这里我们使用了 lambda 表达式，用刚刚创建的 `dis` 对象生成随机数，然后我们使用 `fill_data` 函数填充数据。

接着检查 `Order` 枚举变量，如果是升序，我们使用标准可的排序算法进行排序；如果是降序，我们增加一个比较函数 `std::greater<int>()`，然后排序。

返回的 `arr` 是一个生成的整数序列。

对于浮点数，只要把整数的正态分布对象改成浮点数的即可，即 `std::uniform_real_distribution`。

3 进行测试

下面是我们的测试代码:

```
int main() {
    long long size = 1e7 + 5;

    // 测试不同排列组合
    std::vector<std::pair<Order, Repeat>> combinations = {
        {Order::ASC, Repeat::NO},
        {Order::ASC, Repeat::YES},
        {Order::DESC, Repeat::NO},
        {Order::DESC, Repeat::YES},
        {Order::RANDOM, Repeat::NO},
        {Order::RANDOM, Repeat::YES}
    };

    for (const auto& combination : combinations) {
        Order order = combination.first;
        Repeat repeat = combination.second;
```

```
        std::cout << "Testing with Order: "
                  << (order == Order::ASC ? "ASC" : (order == Order::DESC ? "DESC" : "RANDOM"))
                  << ", Repeat: "
                  << (repeat == Repeat::NO ? "NO" : "YES")
                  << std::endl;

        std::cout << "Test integer:" << '\n';
        std::vector<int> data_int = generate_data_int(size, order, repeat);
        compare(data_int);

        std::cout << "Test double:" << '\n';
        std::vector<double> data_double = generate_data_double(size, order, repeat);
        compare(data_double);
    }

    return 0;
}
```

作业要求至少 1e6 的数据，这里我们使用了 1e7+5 的数据量。
我们对六种排列组合进行测试，每种情况下，我们生成整数和浮点数序列，每次测试都有对应结果。
我们使用 `clang++ test.cpp -o test -std=c++20 -O2` 来进行编译，然后运行`./test` 进行测试。
完成测试总共大概需要一分半到两分钟。
以下是某一次的测试输出:

```
Testing with Order: ASC, Repeat: NO
Test integer:
My Heap Sort: Duration: 0.5548 seconds
STL Heap Sort: Duration: 0.4965 seconds
Two sorting algorithms are correct.

Test double:
My Heap Sort: Duration: 0.6206 seconds
STL Heap Sort: Duration: 0.4936 seconds
Two sorting algorithms are correct.

Testing with Order: ASC, Repeat: YES
Test integer:
My Heap Sort: Duration: 0.5784 seconds
STL Heap Sort: Duration: 0.4909 seconds
Two sorting algorithms are correct.

Test double:
My Heap Sort: Duration: 0.6434 seconds
STL Heap Sort: Duration: 0.4913 seconds
Two sorting algorithms are correct.
```

```
Testing with Order: DESC, Repeat: NO
Test integer:
My Heap Sort: Duration: 0.6010 seconds
STL Heap Sort: Duration: 0.5742 seconds
Two sorting algorithms are correct.

Test double:
My Heap Sort: Duration: 0.6939 seconds
STL Heap Sort: Duration: 0.6015 seconds
Two sorting algorithms are correct.

Testing with Order: DESC, Repeat: YES
Test integer:
My Heap Sort: Duration: 0.6129 seconds
STL Heap Sort: Duration: 0.6417 seconds
Two sorting algorithms are correct.

Test double:
My Heap Sort: Duration: 0.6897 seconds
STL Heap Sort: Duration: 0.6178 seconds
Two sorting algorithms are correct.

Testing with Order: RANDOM, Repeat: NO
Test integer:
My Heap Sort: Duration: 2.0687 seconds
STL Heap Sort: Duration: 2.0665 seconds
Two sorting algorithms are correct.

Test double:
My Heap Sort: Duration: 2.0905 seconds
STL Heap Sort: Duration: 1.9466 seconds
Two sorting algorithms are correct.

Testing with Order: RANDOM, Repeat: YES
Test integer:
My Heap Sort: Duration: 1.8029 seconds
STL Heap Sort: Duration: 1.6197 seconds
Two sorting algorithms are correct.

Test double:
My Heap Sort: Duration: 2.0983 seconds
STL Heap Sort: Duration: 1.9387 seconds
Two sorting algorithms are correct.
```

下面是时间对比表格:

数据类型	我的堆排序时间 (秒)	STL 堆排序时间 (秒)
Order: ASC, Repeat: NO		
整数	0.5548	0.4965
浮点数	0.6206	0.4936
Order: ASC, Repeat: YES		
整数	0.5784	0.4909
浮点数	0.6434	0.4913
Order: DESC, Repeat: NO		
整数	0.6010	0.5742
浮点数	0.6939	0.6015
Order: DESC, Repeat: YES		
整数	0.6129	0.6417
浮点数	0.6897	0.6178
Order: RANDOM, Repeat: NO		
整数	2.0687	2.0665
浮点数	2.0905	1.9466
Order: RANDOM, Repeat: YES		
整数	1.8029	1.6197
浮点数	2.0983	1.9387

表 1: 不同排列组合下的排序时间

可以看到，首先我们的排序算法是正确的。在开启 O2 优化的情况下，我们的堆排序算法的效率和 STL 堆排序算法的效率相差不大，标准库的排序稍微快一些。浮点数的时间比整数稍微长一点，原因可能是浮点数据类型的比较和交换操作比整数类型的操作耗时更长。

关于我们的堆排序和 STL 堆排序的时间差异，可能的原因包括

- **实现细节**：STL 的 `std::make_heap` 和 `std::sort_heap` 是经过高度优化的通用算法，适用于各种类型的数据和比较函数。它们在实现上可能包含了一些我们自定义实现中没有的优化。
- **内存访问模式**：STL 算法可能在内存访问模式上进行了优化，减少了缓存未命中（cache miss）的情况，从而提高了性能。
- **编译器优化**：编译器在优化 STL 库代码时，可能会进行一些特定的优化，而这些优化在我们自定义实现的代码中可能无法完全发挥作用。值得一提的是，在不开启优化的情况下，STL 堆排序的性能会比我们自定义实现的堆排序更差。
- **算法复杂度**：虽然堆排序的时间复杂度在最坏情况下是 $O(n \log n)$，但常数因子也会影响实际运行时间。STL 实现可能在常数因子上进行了优化。