1 引言

堆排序（HeapSort）是一种基于堆结构的高效排序算法。通过构建大顶堆或小顶堆，并反复执行堆化操作，可以实现时间复杂度为 $O(n \log n)$ 的排序。该报告详细记录了本人堆排序的实现过程、测试方法、性能对比以及算法分析。

2 算法描述

2.1 整体思路

本人对于堆排序设计的思路如下：

1. 将输入数组构建为**大顶堆**，此时最大元素位于堆顶。
2. 将堆顶元素与堆尾元素交换，将对顶元素（最大元素）放在了数组的末尾，实现了单个元素的正确排序。
3. 对堆顶元素执行堆化操作（`siftDown`），恢复堆的性质。
4. 重复上述步骤，每一次循环可以将单个元素正确排序，而直至堆的未排序长度变为 1 的时候，就完成了从小到大的排序。

2.2 必要函数的功能

本次实现的堆排序主要包括以下两个函数：

- `siftDown`: 用于堆化操作。它从节点 i 开始，从顶至底进行堆化操作，恢复堆的性质。
- `heapSort`: 堆排序的主函数，首先完成建堆操作，从最后一个非叶结点开始向前遍历至根节点，完成堆化操作；然后交换根节点和最右叶节点，并以根节点为起点从顶至底进行堆化，然后循环这样的过程最终完成堆排序。

2.3 实现细节

在实现中，我采取了如下策略：

1. **构建大顶堆：**从最后一个非叶节点开始向上进行堆化操作，时间复杂度为 $O(n)$。
2. **堆化操作：**利用 `siftDown` 函数对当前堆顶元素进行调整，直到满足堆的性质。
3. **原地排序：**在堆排序中，不需要额外的存储空间，直接在输入数组上完成操作。

3 测试流程

3.1 测试方法

1. 随机生成 4 种长度为 10^6 的测试序列，包括随机序列、有序序列、逆序序列和部分重复序列。
2. 调用自实现的 `heapSort` 和标准库的 `std::sort_heap()` 对各序列分别进行排序。
3. 使用 `chrono` 记录每次排序的时间，并验证排序结果的正确性。
4. 通过 Valgrind 工具检测内存泄漏，确保程序运行稳定。

4 性能分析

4.1 测试概述

按要求针对以下四种序列对堆排序进行了测试：

1. **随机序列**：元素为随机生成的整数。
2. **有序序列**：元素已经按升序排列。
3. **逆序序列**：元素按降序排列。
4. **部分重复序列**：包含大量重复元素。

所有序列长度均为 10^6，并与标准库的 `std::sort_heap()` 进行了性能对比。

4.2 测试结果

以下是堆排序与 `std::sort_heap()` 的性能测试结果（单位：秒）：

序列类型	堆排序时间 (s)	std::sort_heap 时间 (s)
随机序列	0.538	0.310
有序序列	0.427	0.249
逆序序列	0.457	0.271
部分重复序列	0.533	0.336

表 1: 堆排序与 std::sort_heap 的性能对比

5 时间复杂度与效率差异分析

5.1 时间复杂度分析

堆排序的时间复杂度分为两部分：

- **构建大顶堆：**对每个非叶节点执行堆化操作，总复杂度为 $O(n)$。
- **堆化与排序：**从堆中提取最大元素的时间复杂度为 O(logn)，共循环 n-1 轮，故提取最大元素（包括堆化操作）的时间复杂度为 $O(nlogn)$。

所以总的时间复杂度为 $O(n) + O(nlogn) = O(nlogn)$。

5.2 与 `std::sort_heap()` 的效率差异

从测试结果可以看出，标准库的 `std::sort_heap()` 明显快于我实现的 `heapSort`，我个人认为原因如下：

1. **底层优化：**标准库的实现经过高度优化，可能使用了硬件加速或并行化操作，而我的实现只针对通用性，没有针对性能进行优化。
2. **缓存命中率：**标准库对内存访问模式进行了优化，减少了缓存未命中，而我的实现在交换和堆化操作中可能导致更多的缓存未命中。
3. **函数调用开销：**我的实现中使用了较多的递归或循环，可能带来额外的函数调用开销。

6 结论

我实现的堆排序性能较标准库的 `std::sort_heap()` 有一定的性能差异，但时间复杂度符合理论预期，且能够正确处理多种类型的输入序列，感觉还行（手动狗头）