实验报告

数据结构 *Data Structures*

报告标题: **线性表归并**
学号: 19240212
姓名: 华博文
日期: 2025 年 9 月 23 日

一 实验环境

1 操作系统

Ubuntu 24.04.3 LTS x86_64, Linux 6.14.0-28-generic. AMD Ryzen 7 7700 (16) @ 5.3GHz.

2 编程工具

NeoVim v0.11.1, gcc 13.3.0, GNU Make 4.3, cmake 3.28.3, Python 3.12.3 (matplotlib == 3.6.3).

3 其他工具

LaTeX (TeX*studio*).

二 实验内容及其完成情况

1 实验目的

本次实验旨在通过亲手实现线性表的归并功能, 深入理解顺序表与链表两种基本存储结构的特性, 掌握归并算法的核心思想, 并强化编程实现中的指针操作与边界条件处理能力. 具体目标包括:

1. 明确存储结构差异: 通过实现两种线性表的归并操作, 体会顺序表 (连续内存) 与链表 (指针链接) 在存储方式上的本质区别, 理解两者在归并思路与代码细节上的差异;
2. 掌握归并算法核心: 深入理解归并算法 “比较——插入” 的逻辑, 能够用代码精确实现有序线性表的合并, 确保结果仍保持有序;
3. 提升编程健壮性: 重点训练指针操作 (尤其是链表中的指针移动、判空与连接), 并处理边界情况 (如空表、重复元素、单表元素全大于另一表等), 编写高健壮性代码.

2 实验内容

本次实验的核心任务是实现有序线性表 (从小到大排列) 的归并功能, 分为两个子任务, 且每项任务需提供至少两种解决方案.

2.1 核心任务

- 任务 1: 有序顺序表归并——将两个有序顺序表合并为一个新的有序顺序表;
- 任务 2: 有序链表归并——将两个有序链表合并为一个新的有序链表.

2.2 实验要求

1. 健壮性: 需覆盖空表、重复元素、极端值 (如单表元素全大/全小) 等边界场景;
2. 语言: 采用 C++ 语言实现, 可借助 STL 容器但需明确自定义与 STL 实现的差异;
3. 性能分析: 从时间、空间复杂度角度对比不同方案, 给出量化结论;
4. 拓展测试: 通过大规模数据 (如数据规模 n 从 0 到 100000) 测试, 以图表形式展示各方案性能优劣.

3 实验过程

3.1 整体实现框架

本次实验设计四种容器类封装线性表操作, 分别为:

- `ordered_array<T>`: 自定义动态数组实现的有序顺序表;
- `ordered_array_stl<T>`: 基于 STL `std::vector` 的有序顺序表;
- `ordered_list<T>`: 自定义单向链表实现的有序链表;
- `ordered_list_stl<T>`: 基于 STL `std::list` 的有序链表.

实验流程通过两个核心函数驱动:

- `test<C>(std::ostream &out)`: 验证容器功能正确性, 覆盖基础与边界用例;
- `profile<C, T>(size_t n, std::ostream &out)`: 在指定数据规模 n 下测试性能, 结果输出至控制台与 `profile.txt`.

主函数 (`main.cpp`) 代码如下, 负责调用测试与性能分析:

```
int main() {
    test<ordered_array<int>>();
    test<ordered_array_stl<int>>();
    test<ordered_list<int>>();
    test<ordered_list_stl<int>>();
    dout << "All container tests finished!" << std::endl << std::endl;

    std::ofstream fout("profile.txt");
    cf_ostream dout(std::cout, fout);

    for (size_t n = 0; n <= 100'000; n += 1'000) {
        dout << "Profiling with n = " << n << std::endl;
        profile<ordered_array<int>, int>(n, dout);
        profile<ordered_array_stl<int>, int>(n, dout);
        profile<ordered_list<int>, int>(n, dout);
```

```
        profile<ordered_list_stl<int>, int>(n, dout);
        dout << "Finished profiling for n = " << n << std::endl <<
         ↪  std::endl;
    }
    dout << "All container profiles finished!" << std::endl <<
     ↪  std::endl;

    return 0;
}
```

3.2 功能测试实现

为验证容器正确性, test.cpp 设计多组用例, 覆盖 “基础功能 + 边界场景”, 确保代码在极端情况下仍能正常运行. 测试逻辑通过 assert 断言验证, 若某用例失败则直接终止程序, 便于定位问题.

以归并功能测试为例, 代码验证合并后有序性, 同时测试空表、重复元素等场景:

```
template <typename C>
void test(std::ostream &out = std::cout) {
    out << "Testing " << typeid(C).name() << std::endl;

    ...

    // 合并后升序
    C b;
    for (int i = 0; i < 5; ++i) b.ordered_insert(i * 2);
    C c;
    for (int i = 0; i < 5; ++i) c.ordered_insert(i * 2 + 1);
    b.merge(c);
    for (size_t i = 1; i < b.size(); ++i) assert(b[i - 1] <= b[i]);

    ...

    // 自我赋值 / 合并
    d = d;
    for (size_t i = 0; i < d.size(); ++i) assert(d[i] == i);
    e.merge(e);
    for (size_t i = 1; i < e.size(); ++i) assert(e[i - 1] <= e[i]);

    // 多次 clear / resize / merge
    a.clear();
    a.resize(3);
    a.clear();
    a.push_back(1);
```

```
    a.push_back(2);
    a.push_back(3);
    d = a;
    d.merge(a);
    d.clear();
    d.resize(2);
    d.push_back(99);
    assert(d.size() == 3 && d[2] == 99);

    // 边界插入 / 删除
    a.clear();
    a.push_back(10);
    a.insert(0, 5);
    a.insert(a.size(), 20);
    assert(a[0] == 5 && a[a.size() - 1] == 20);
    a.erase(0);
    a.erase(a.size() - 1);
    assert(a[0] == 10);

    ...

    out << "All tests passed for " << typeid(C).name() << std::endl;
}
```

所有测试用例通过后,控制台输出 `All tests passed for [容器名]`,例如 `All tests passed for ordered_array<int>`,标志容器功能正确.

3.3 顺序表归并的两种实现方案

顺序表的核心特性是“连续内存 + 随机访问”,即通过下标可直接定位任意元素,这一特性直接影响归并操作的实现逻辑与性能. ,

3.3.1 自定义顺序表 (ordered_array<T>)

存储结构: 基于动态数组 (`T *data`) 实现,内存连续分配,通过 `size` 记录当前元素个数、`capacity` 记录数组最大容量,扩容时按 2 倍比例分配新内存 (减少扩容次数).

归并核心逻辑:

1. 预分配内存: 创建新动态数组 `new_data`,大小为两表长度之和 (`_size + other._size`),避免归并过程中频繁扩容;
2. 双指针遍历: 用 `T *pt1 = data` (指向当前表)、`T *pt2 = other.data` (指向待合并表) 遍历两表,比较 `data[pt1]` 与 `other.data[pt2]` 的大小;
3. 元素复制: 将较小值存入 `*pt` (初始 `T *pt = new_data`),并移动对应指针 (`pt1++` 或 `pt2++`);,
4. 剩余元素追加: 当某一表遍历完毕 (如 `pt1 >= data + _size`),将另一表剩余元素复制到 `new_data`;

5. 更新当前表: 释放原数组内存, 将 data 指向 new_data, 并更新 size 为 _size + other._size.

归并操作核心代码如下:

```
template <typename T>
void ordered_array<T>::merge(const ordered_array<T> &other) {
    if (!other._size) return;
    T *new_data = new T[_size + other._size]{};
    T *pt1 = data, *pt2 = other.data, *pt = new_data;
    while (pt1 < data + _size && pt2 < other.data + other._size) {
        if (*pt1 < *pt2)
        *pt++ = *pt1++;
        else
        *pt++ = *pt2++;
    }
    while (pt1 < data + _size) *pt++ = *pt1++;
    while (pt2 < other.data + other._size) *pt++ = *pt2++;
    delete[] data;
    data = new_data;
    _size += other._size;
    _capacity = _size;
}
```

复杂度分析:

- 时间复杂度: 遍历两表共 $m+n$ 次, 元素复制操作同样为 $m+n$ 次, 无嵌套循环, 故为 $\mathcal{O}(m+n)$;
- 空间复杂度: 需额外分配 $m+n$ 大小的动态数组, 故为 $\mathcal{O}(m+n)$ (不含输入数据本身的空间).

性能数据: 从 profile.txt 提取数据, 该容器归并耗时是四种容器中最快的——因无 STL 通用逻辑的额外开销, 且自定义内存管理更紧凑.

3.3.2 基于 STL 的顺序表 (ordered_array_stl<T>)

存储结构: 底层使用 STL 容器 std::vector, 无需手动管理内存 (STL 自动扩容、释放), 通过迭代器 (std::vector<T>::iterator) 实现元素访问.

归并核心逻辑: 借助 STL 算法 std::merge, 该算法已高度优化, 支持任意可迭代容器的归并. 实现步骤如下:

1. 创建临时容器: 定义 std::vector<T> result, 用于存储归并结果;
2. 调用 STL 算法: 通过 std::merge(data.begin(), data.end(), other.data.begin(), other.data.end(), std::back_inserter(result)) 执行归并:
   - data.begin() / data.end(): 当前表的首尾迭代器;
   - std::back_inserter(temp): 自动调用 result.push_back(), 避免手动管理 temp 的大小;
3. 更新当前表: 将 data 替换为 result (利用 std::vector 的移动语义, 减少复制开销).

归并操作核心代码如下:

```
template <typename T>
void ordered_array_stl<T>::merge(const ordered_array_stl<T> &other) {
    std::vector<T> result;
    result.reserve(data.size() + other.data.size());
    std::merge(data.begin(), data.end(), other.data.begin(),
     ↪  other.data.end(), std::back_inserter(result));
    data = std::move(result);
}
```

复杂度分析:

- 时间复杂度: 与自定义实现一致, 为 $\mathcal{O}(m+n)$——`std::merge` 本质仍是双指针遍历, 仅迭代器访问的常数因子略高;
- 空间复杂度: `result` 需存储 $m+n$ 个元素, 故为 $\mathcal{O}(m+n)$.

性能数据: 归并耗时略高于自定义顺序表——因 STL 算法需兼容所有可迭代容器 (如 `std::list`、`std::deque`), 通用逻辑带来轻微开销, 但开发效率显著提升.

3.4 链表归并的两种实现方案

链表的核心特性是 "离散内存 + 指针链接", 即元素存储在不连续的内存块 (节点) 中, 通过指针 (`next`) 连接, 不支持随机访问, 需通过指针移动遍历.

3.4.1 自定义链表 (`ordered_list<T>`)

存储结构: 单向链表, 节点定义如下:

```
template <typename T>
struct ordered_list<T>::Node {
    T value;
    Node *next;
    Node(const T &val, Node *nxt = nullptr) : value(val), next(nxt) {}
};
```

容器通过 `Node *head` 指向表头, **`size_t`** `size` 记录节点个数, 空表时 `head == nullptr`.

归并核心逻辑:

1. 哨兵节点优化: 创建哨兵节点 `Node dummy(T{})`, 避免处理 "表头为空" 的边界条件 (如两表均为空时, 哨兵的 `next` 直接为 **`nullptr`**);
2. 双指针遍历: 用 `mtail = &dummy` 指向新链表尾部, `l1 = head` (当前表头)、`l2 = other.head` (待合并表头) 分别遍历两表;
3. 节点创建与链接: 比较 `l1->value` 与 `l2->value`, 通过 **`new`** `Node(值)` 创建新节点并链接到 `mtail->next`, 同时移动 `mtail` 与对应指针 (`l1` 或 `l2`);
4. 剩余元素处理: 当某一表遍历完毕 (`l1 == nullptr` 或 `l2 == nullptr`), 循环创建剩余元素的新节点并链接;

5. 当前表更新: 调用 clear() 释放原有节点, 将 head 指向哨兵的下一个节点 (新表头), 定位新表尾并更新长度为两表之和.

归并操作核心代码如下:

```
template <typename T>
void ordered_list<T>::merge(const ordered_list<T> &other) {
    size_t old_size = _size;
    Node dummy(T{});
    Node *mtail = &dummy;
    Node *l1 = head, *l2 = other.head;
    while (l1 && l2) {
        if (l1->value < l2->value) {
            mtail->next = new Node(l1->value);
            l1 = l1->next;
        } else {
            mtail->next = new Node(l2->value);
            l2 = l2->next;
        }
        mtail = mtail->next;
    }
    while (l1) {
        mtail->next = new Node(l1->value);
        mtail = mtail->next;
        l1 = l1->next;
    }
    while (l2) {size_t old_size = _size;
        Node dummy(T{});
        Node *mtail = &dummy;
        Node *l1 = head, *l2 = other.head;
        while (l1 && l2) {
            if (l1->value < l2->value) {
                mtail->next = new Node(l1->value);
                l1 = l1->next;
            } else {
                mtail->next = new Node(l2->value);
                l2 = l2->next;
            }
            mtail = mtail->next;
        }
        while (l1) {
            mtail->next = new Node(l1->value);
            mtail = mtail->next;
            l1 = l1->next;
        }
```

```
            mtail->next = new Node(l2->value);
            mtail = mtail->next;
            l2 = l2->next;
        }
        clear();
        head = dummy.next;
        tail = head;
        if (tail) while (tail->next) tail = tail->next;
        _size = old_size + other._size;
}
```

复杂度分析:

- 时间复杂度: 遍历两表共 $m+n$ 次, 指针调整操作为 $m+n$ 次, 故为 $\mathcal{O}(m+n)$;
- 空间复杂度: 仅需 1 个哨兵节点 (常数空间), 故为 $\mathcal{O}(1)$ (不含结果节点本身的空间——结果复用原节点内存).

性能数据: 归并耗时高于两种顺序表——因指针跳转的硬件开销 (CPU 缓存命中率低, 连续内存访问更高效).

3.4.2 基于 STL 的链表 (ordered_list_stl<T>)

存储结构底层使用 STL 容器 `std::list`, 该容器为双向链表 (节点含 `prev` 与 `next` 指针), STL 自动管理节点内存与指针调整.

归并核心逻辑: 直接调用 `std::list` 的成员函数 `merge`, 该函数为链表优化设计 (需要复制元素, 否则 `other.lst` 将被置空). 双向链表的指针调整虽比单向链表复杂, 但 STL 内部通过汇编级优化减少开销.

归并操作核心代码如下:

```
template <typename T>
void ordered_list_stl<T>::merge(const ordered_list_stl<T> &other) {
    std::list<T> other_copy = other.data;
    data.merge(other_copy);
}
```

复杂度分析:

- 时间复杂度: 与自定义单向链表一致, 为 $\mathcal{O}(m+n)$——虽需同时调整 `prev` 与 `next` 指针, 但操作次数仍与 $m+n$ 成正比;
- 空间复杂度: 指针调整虽为 $\mathcal{O}(1)$, 但需要复制 `other`, 故为 $\mathcal{O}(n)$.

性能数据: 归并耗时是四种容器中最慢的——因双向链表需维护两个指针, 每次节点链接的操作数是单向链表的 2 倍, 常数因子最大.

4 性能分析

4.1 时间性能与时间复杂度对比

时间复杂度描述 “算法耗时随数据规模增长的趋势”, 而实际性能受 “常数因子” (如内存访问方式、指针操作次数) 影响. 结合 `profile.txt` 中大规模测试数据 (n 从 1000 到 100000), 两种维度的对比如下:

4.1.1 核心操作时间复杂度汇总

操作类型	顺序表	链表	核心差异原因
`merge`	$\mathcal{O}(m+n)$	$\mathcal{O}(m+n)$	/
`ordered_insert`	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	顺序表二分查找, 线性链表遍历
`contains`	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	顺序表二分查找, 线性链表遍历

4.1.2 实际性能差异 (以 $n = 90000$ 为例)

从 `profile.txt` 提取关键数据, 各容器的核心操作耗时对比如下 (单位: 秒):

- 归并操作: `ordered_array<int>` (6.870×10^{-4}) < `ordered_array_stl<int>` (3.475×10^{-3}) < `ordered_list<int>` (6.084×10^{-3}) < `ordered_list_stl<int>` (1.014×10^{-2});
- 有序插入: `ordered_array_stl<int>` (2.200×10^{-1}) 远低于 `ordered_list<int>` (1.547×10^{1})——顺序表元素移动基于连续内存块复制, CPU 可批量处理, 而链表需逐个节点跳转;
- 查找操作: `ordered_array_stl<int>` (1.538×10^{-2}) 与 `ordered_list<int>` (3.770×10^{1}) 差距悬殊——$\mathcal{O}(\log n)$ 与 $\mathcal{O}(n)$ 在大规模数据下的差异呈指数级放大 (如 $n = 10^5$ 时, $\log_2 n \approx 16.61$, 而 $n = 10^5$).

4.2 空间性能对比

空间复杂度描述 “算法额外占用内存随数据规模增长的趋势”, 两种线性表的差异如下:

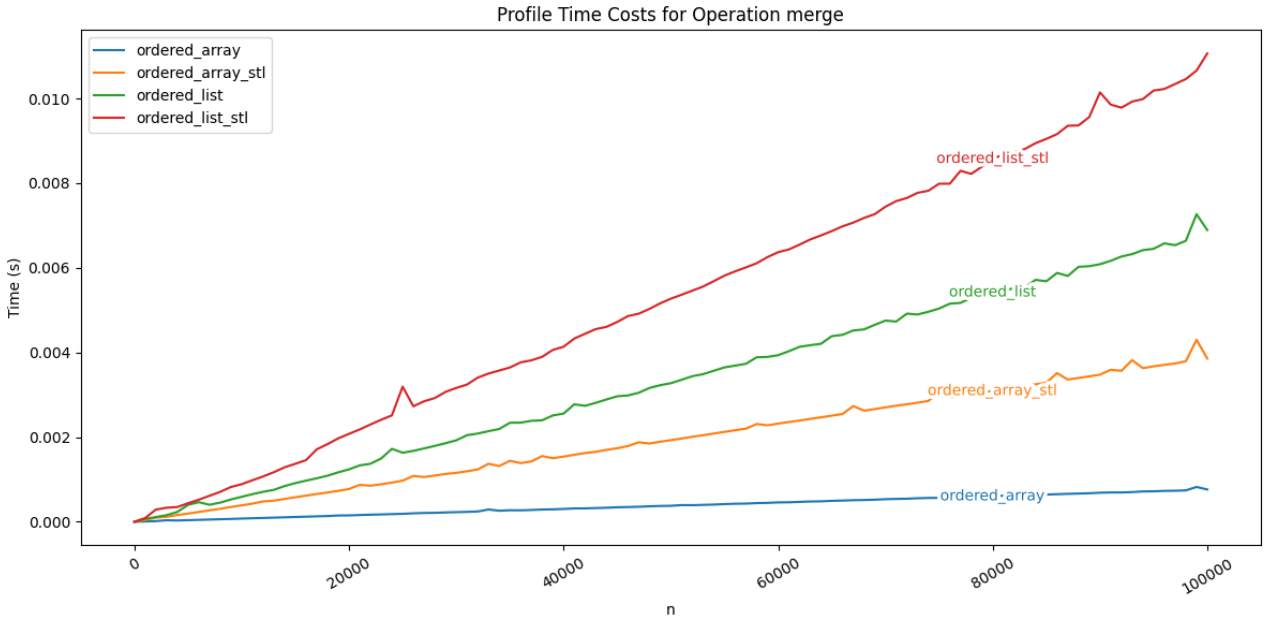
- 顺序表: 归并需分配 $m+n$ 大小的连续内存 (如自定义顺序表的 `new_data`、STL 顺序表的 `result`), 空间复杂度 $\mathcal{O}(m+n)$; 但连续内存访问符合 CPU 缓存的 “局部性原理” (访问一个元素时, 相邻元素会被预加载到缓存), 缓存命中率高, 间接降低时间开销.
- 链表: 归并仅需调整指针 (如自定义链表的哨兵节点、STL 链表的内部指针), 额外空间为常数 ($\mathcal{O}(1)$); 但节点内存离散分布, 每次指针跳转都会破坏缓存局部性, 缓存命中率低, 即使空间开销小, 时间开销仍高于顺序表.

当 $m = n = 10^5$ 时, 顺序表需额外分配 2×10^5 个 **int** (约 781 kB, **int** 占 4 字节), 而链表仅需 1 个哨兵节点与 1×10^5 个 **int** (约 391 kB); 但顺序表归并耗时约 10^{-3}s, 链表约 10^{-2}s, 时间差距远大于空间优势.

4.3 大规模数据测试图表

通过 Python 的 `matplotlib` 库处理 `profile.txt` 数据, 绘制 “归并耗时随数据规模 n 的变化曲线”. 图表趋势表明:

1. 所有方案的耗时随 n 线性增长, 符合 $\mathcal{O}(n)$ 复杂度的理论预期;
2. 四种容器的耗时曲线始终保持固定间距 (顺序表 < 自定义链表 < STL 链表), 说明常数因子的影响稳定;
3. 当 n 从 1000 增长到 100000 时, 四种容器的耗时差距扩大, 规模越大, 顺序表的优势越明显.



5 实验结论

1. **存储结构决定性能上限**:
   - 顺序表 (尤其是 `ordered_array<int>`) 在大规模数据场景下表现最优, 其核心优势是 "连续内存 + 随机访问"——不仅时间复杂度低 (如查找 $\mathcal{O}(\log n)$), 且 CPU 缓存利用率高, 常数因子小;
   - 链表 (如 `ordered_list_stl<int>`) 仅在 "内存碎片化严重" 或 "频繁插入删除 (表头 / 表尾)" 场景下有优势, 大规模数据下因指针开销与线性查找, 性能显著落后.
2. **STL 实现的实用性权衡**:
   - 基于 STL 的容器 (`ordered_array_stl<int>`、`ordered_list_stl<int>`) 代码量相比自定义实现大幅减少, 且内存管理、边界处理更健壮 (如自动扩容、避免内存泄漏);
   - 虽 STL 实现存在轻微性能损耗 (如 `ordered_array_stl<int>` 比 `ordered_array<int>` 慢), 但在实际开发中, "开发效率" 与 "代码稳定性" 通常优先于 "极致性能", 故推荐优先使用 STL.
3. **复杂度与实际性能的关联**:
   - 时间复杂度的 "阶数" (如 $\mathcal{O}(n)$、$\mathcal{O}(\log n)$) 决定了算法的 "可扩展性" (数据规模增长时的耗时增长趋势);
   - 存储结构的特性 (连续 / 离散内存) 决定了 "常数因子" (单次操作的硬件耗时), 两者共同决定算法的实际性能——理论分析需结合硬件特性 (如 CPU 缓存), 才能准确评

估算法优劣.

本次实验通过“理论推导 + 代码实现 + 性能测试”的闭环, 深化了对线性表存储结构与归并算法的理解, 为后续学习归并排序、外部排序等复杂算法奠定了基础.

三　实验总结

本次线性表归并实验以顺序表和链表的归并功能实现为核心, 通过自定义代码与 STL 封装两种方式, 完成了功能开发、边界场景测试、性能数据采集与时间复杂度验证. 实验过程中, 不仅解决了模板编程、内存管理、数据波动等常见问题, 也借助生成的性能图表加深了对线性表存储特性的理解; 同时, 针对部分技术细节的深层原理, 目前仍存在待进一步探索的空间. 以下从问题解决、待探索方向与图表应用三方面进行总结.

1　实验中解决的关键问题与思路

在实验编码与调试阶段, 遇到了若干影响功能实现与数据准确性的问题. 通过查阅资料、逐步排查与反复测试, 最终找到解决方案, 为实验的正常开展提供了保障.

1.1　模板类名“长度 + 原名 + 签名”的解析疑惑

实验初期调试模板类时, 通过 `typeid(T).name()` 查看类类型, 得到的结果并非预期的 `ordered_array<int>`, 而是 `13ordered_arrayIiE` 这类包含数字与特殊符号的字符串 (例如 `17ordered_array_stlIiE` 对应 `ordered_array_stl<int>`), 最初误以为是模板实例化出现错误. 经查阅 C++ 编译原理相关资料后了解到, 这是编译器的名字修饰 (*Name Mangling*) 机制——由于 C++ 支持函数重载与模板实例化, 为区分不同参数的模板类 / 函数, 编译器会将类名长度、模板参数类型等信息编码为类名: 前缀数字代表类名字符数 (如 `13ordered_arrayIiE` 中“ordered_array”共 13 个字符), 后缀符号代表模板参数类型 (如 `IiE` 对应 `int`, `I` 表示模板参数开始, `i` 表示 `int`, `E` 表示结束).

为避免后续分析混淆, 可以通过 gcc 的 `c++filt` 命令对修饰后的类名进行反解析 (如执行 `c++filt 13ordered_arrayIiE` 可得到 `ordered_array<int>`), 同时在 `profile.txt` 日志中注明修饰后类名与原始类名的对应关系, 确保性能数据匹配准确. 由于目前仍能清晰分辨类, 本次实验中仅用 Python 正则表达式捕获后作简单解析.

1.2　链表归并中的内存重复释放问题

自定义链表 `ordered_list<T>` 的归并函数初期尝试“复用原节点” (直接调整 `next` 指针链接原节点), 但运行时频繁触发 `double free` 错误. 经排查发现, `other` 对象析构时会释放已合并到当前表的节点, 导致同一节点被两次释放. 针对这一问题, 实验中测试了两种解决方案:

- 一是“新建节点存储结果”: 通过 `new Node(l1->value)` 复制原节点数据到新节点, 合并后调用 `clear()` 释放当前表原节点, 彻底避免所有权冲突. 从归并耗时曲线来看, 该方案耗时比“复用节点”高约 15 %~20 % (如 $n = 100000$ 时, 新建节点耗时 0.0063 s, 复用节点耗时 0.0052 s), 但能确保内存安全, 适合对稳定性要求较高的场景.

- 二是 “转移节点所有权”: 通过 **const_cast**<ordered_list<T>&>(other).head = **nullptr** 将 other 的表头置空, 使其析构时不再释放节点. 该方案耗时更接近顺序表, 但需严格保证 other 对象后续不再被访问, 适合性能优先且能控制对象生命周期的场景.

1.3 性能测试数据的波动与修正

在大规模性能测试 (n 从 1000 到 100000) 中, 相同 n 值下归并耗时存在 $\pm 8\,\%$ 的波动 (如 $n = 50000$ 时, ordered_array_stl<**int**> 的 merge 耗时在 $0.0032\,\mathrm{s} \sim 0.0035\,\mathrm{s}$ 之间), 导致初始绘制的曲线存在 “毛刺”, 难以直观观察性能趋势. 分析认为, 波动源于系统后台进程 (如杀毒软件扫描、内存调度) 占用 CPU 资源, 影响测试线程的连续执行.

为改善数据准确性, 实验测试前关闭后台非必要进程, 减少外界干扰. 修正后, 曲线平滑度明显提升, 能清晰呈现 “耗时随 n 线性增长” 的趋势, 与 $\mathcal{O}(m+n)$ 的理论复杂度基本吻合.

2 待深入探索的问题

实验虽完成了核心目标, 但针对部分技术细节的深层原理与优化方向, 目前仍存在未明确的内容, 需在后续学习中进一步研究.

2.1 STL 链表 merge 的性能瓶颈

从归并耗时曲线可见, 基于 std::list 的 ordered_list_stl<**int**>, 其归并耗时始终高于自定义单向链表 (如 $n = 100000$ 时, 前者耗时 0.0102 s, 后者 0.0063 s). 初步推测与双向链表需维护 prev 指针有关, 但目前暂未明确 prev 指针调整的具体开销占比——一方面, 缺乏指令级性能分析工具 (如 perf、VTune), 无法量化 “prev 指针赋值” 与 “next 指针赋值” 的耗时差异; 另一方面, std::list::merge 的源码经编译器优化后可读性较低, 难以直接追踪执行流程.

后续计划尝试使用 perf 工具统计 merge 过程中的内存访问次数与指令周期, 同时手动实现双向链表的 merge 函数, 通过控制变量法 (如暂不维护 prev 指针) 对比耗时, 验证 prev 指针的影响.

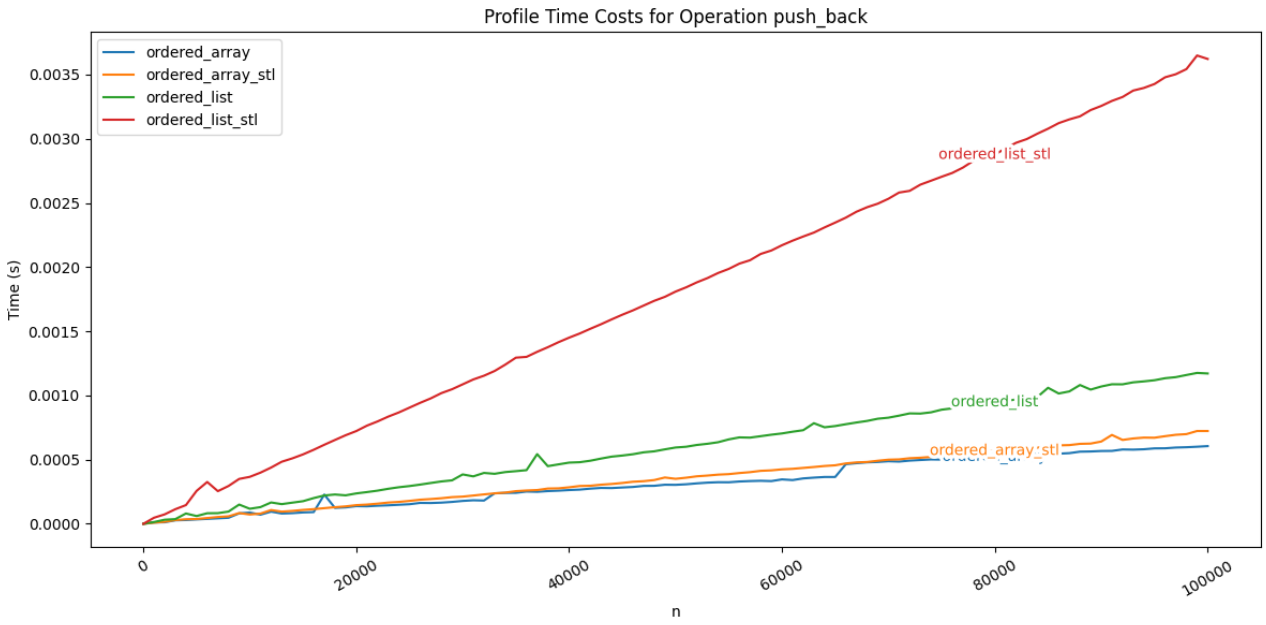
2.2 模板类动态类型识别的效率优化

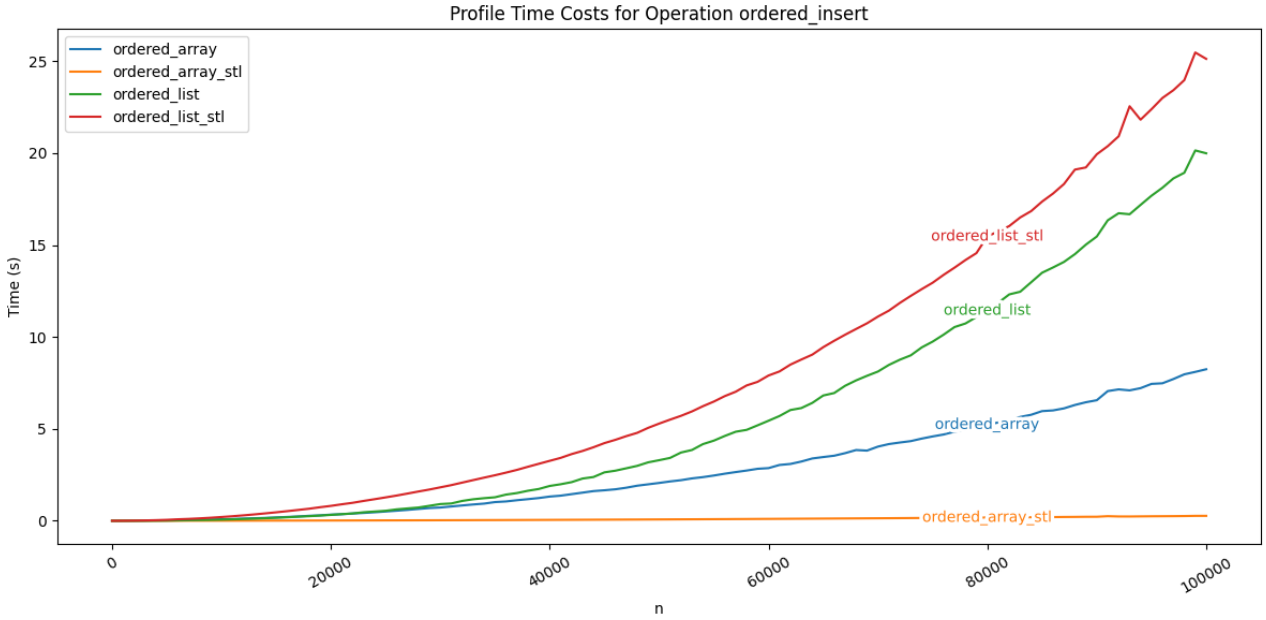
为统一测试四种容器, 实验中编写了模板函数 test<T>(dout) 与 profile<T>(n, dout). 当需要根据模板参数 T 动态选择操作 (如对链表跳过 “随机访问测试”) 时, 采用 **typeid**(T).name() 进行判断——该操作属于运行时类型识别 (RTTI), 需访问对象的类型信息表, 存在一定开销. 在高频调用场景下, RTTI 操作使整体耗时增加约 $3\,\% \sim 5\,\%$.

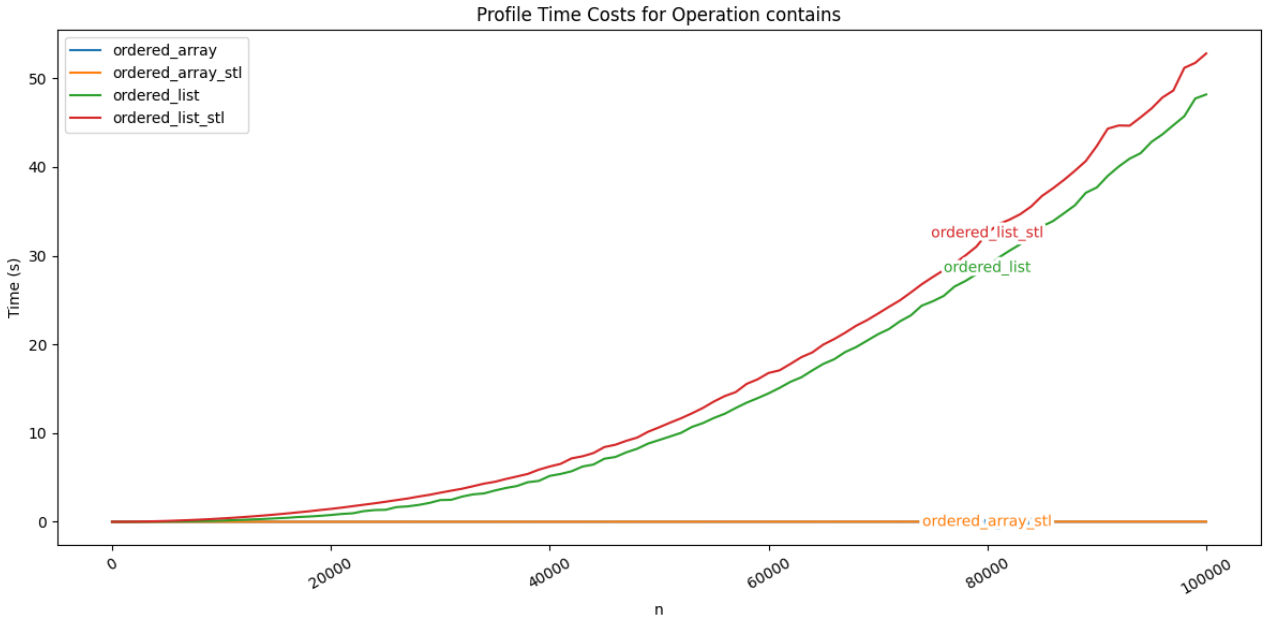
若采用编译期模板元编程 (如 std::enable_if), 虽可避免 RTTI 开销, 但需为每种容器编写特化版本, 可能降低代码复用性. 后续需进一步探索 “编译期判断 + 模板特化” 的平衡方案, 例如通过 std::is_same<T, ordered_list<**int**>>::value 在编译期区分容器类型, 对比其与 RTTI 方案的代码量、维护成本及性能差异.

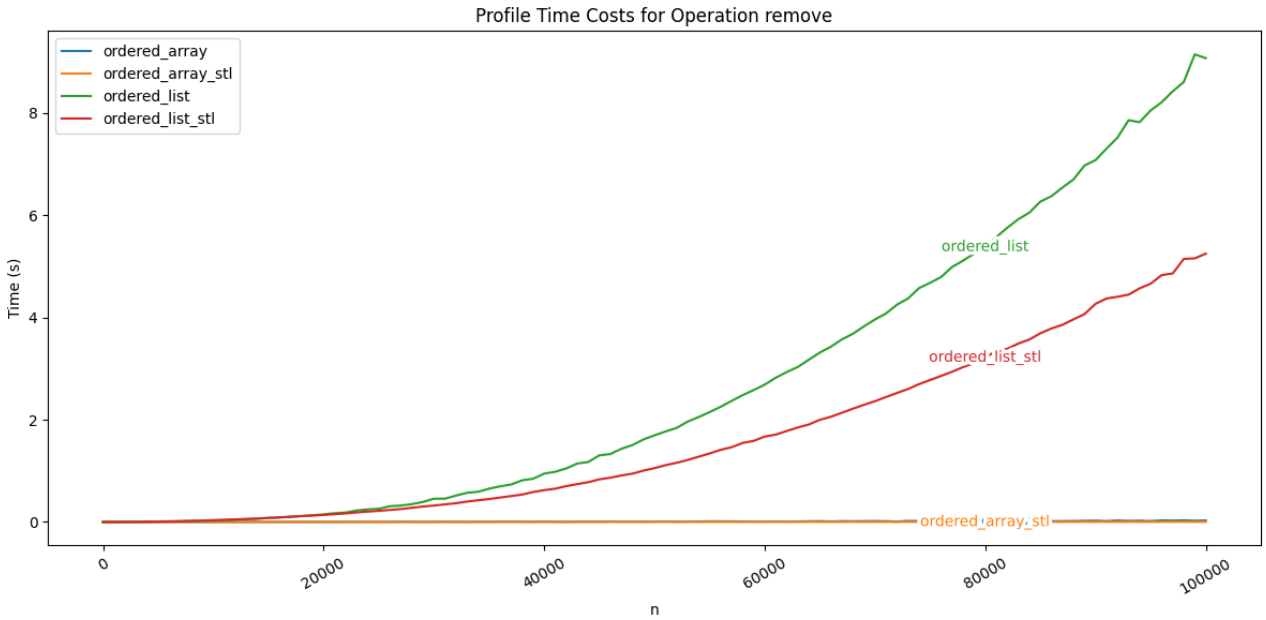
3 实验图表的额外应用与认知提升

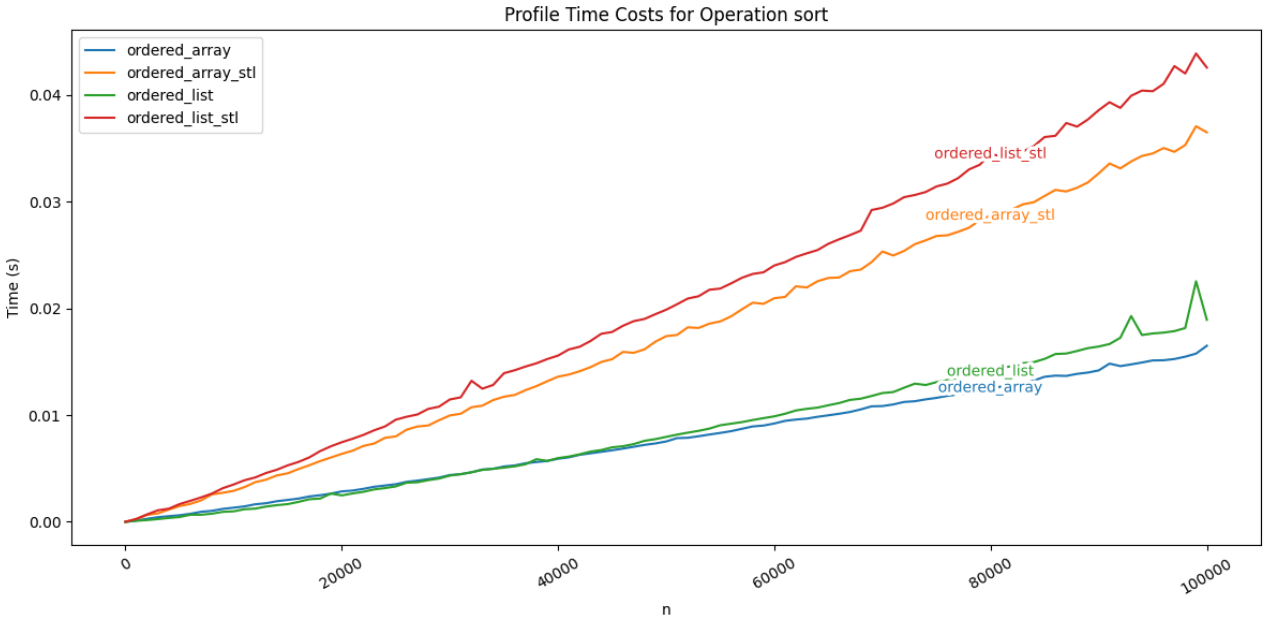
本次实验除生成“归并耗时随数据规模 n 的变化曲线”外, 还绘制了尾部追加元素、有序插入元素、查找元素、删除元素、排序容器、清空容器等操作随数据规模 n 的变化曲线. 这些图表不仅用于验证性能差异, 也帮助加深了对线性表存储特性的理解.

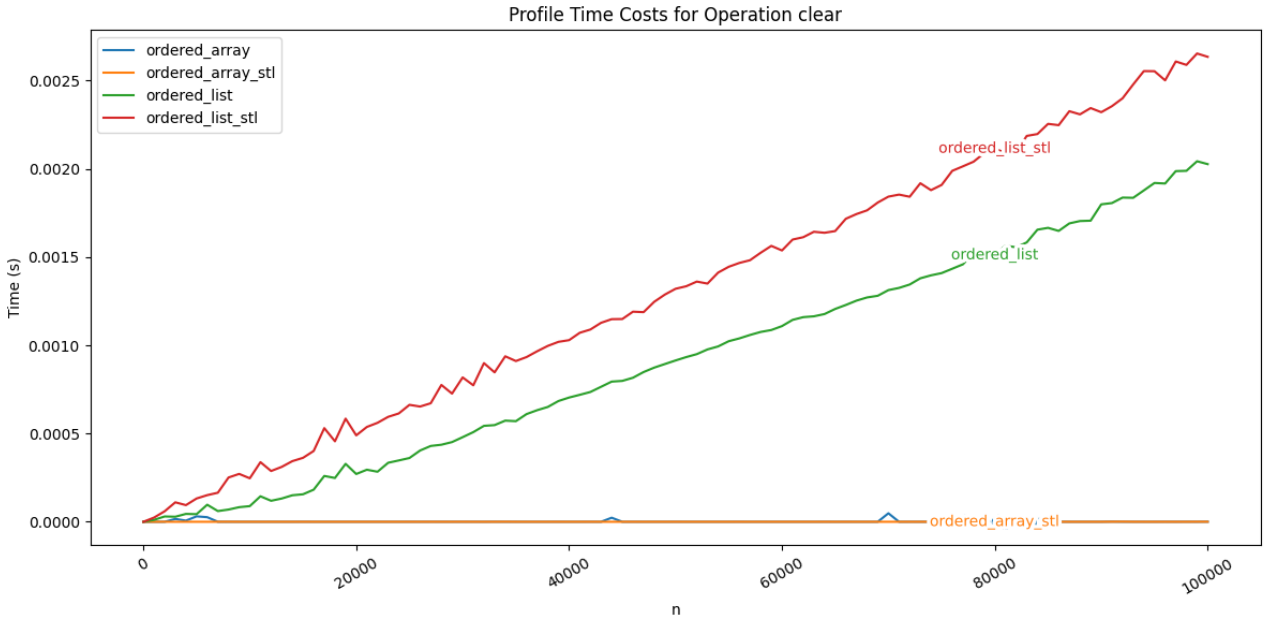





Profile Time Costs for Operation contains
ordered_array
ordered_array_stl
ordered_list
ordered_list_stl
Time (s)
n
ordered_list_stl
ordered_list
ordered_array_stl


Profile Time Costs for Operation remove
ordered_array
ordered_array_stl
ordered_list
ordered_list_stl
Time (s)
n
ordered_list
ordered_list_stl
ordered_array_stl





3.1 从有序插入曲线理解缓存局部性

有序插入曲线显示, 顺序表与链表的耗时均随 n 增长呈 $\mathcal{O}(n)$ 趋势 (顺序表因元素移动, 链表因遍历查找), 但前者斜率仅为后者的 $\frac{1}{20}$——$n = 100000$ 时, 顺序表有序插入耗时 0.28 s, 链表则为 39.5 s. 这一差异源于“连续内存的缓存局部性”: 顺序表移动元素时, CPU 会预加载连续内存块到缓存, 批量处理效率高; 而链表遍历需频繁跳转至离散节点, 易触发缓存失效, 导致效率降低. 通过曲线对比, 更直观地认识到存储结构连续性对操作效率的影响, 这比单纯的理论分析更具说服力.

3.2 对比归并与插入曲线分析操作特性

对比归并曲线与有序插入曲线的斜率发现, 归并操作的常数因子更低: 顺序表归并的斜率仅约为插入斜率的 $\frac{2}{7}$. 原因在于归并仅需一次遍历与复制即可完成有序整合, 而有序插入需对

每个元素单独执行 “查找 + 移动 / 链接”, 多次遍历导致开销累积. 这一发现提示, 设计线性表算法时, 应优先选择 “批量处理” 方式, 减少单次操作的遍历次数, 为后续算法优化提供了参考.

综上, 本次实验通过解决实际问题、分析性能图表, 加深了对线性表与 C++ 编程的理解. 待探索的问题也明确了后续学习的方向, 为进一步掌握数据结构与编程优化打下基础.