平衡树 Rope

> 该部分创作声明

部分内容转载自： 神级STL结构-rope大法（学习笔记），有删改。
可能含有 AIGC 内容，请注意辨别。

声明

rope 最初是作为字符串的存储结构，对**字符类型**有优化 `crope`。但同时他也支持泛型编程（**可比较类型**或小于号**重载**，与上述 `Tree` 相同），一般来说，使用`rope<typename>` 来进行声明，你可以将他理解为一个增强版的 `vector<typename>`。

用法

1. **基本操作**

方法	说明	时间复杂度
`rope<char> r` / `crope r`	构造空 rope	$O(1)$
`crope r("text")`	使用字符串构造	$O(n)$
`r.push_back(c)`	在末尾添加字符	$O(\log n)$
`r.insert(pos, str)`	在位置 pos 插入字符串	$O(\log n + m)$
`r.erase(pos, len)`	从 pos 开始删除 len 个字符	$O(\log n)$
`r.replace(pos, str)`	从 pos 开始替换为字符串	$O(\log n + m)$
`r.substr(pos, len)`	从 pos 开始截取 len 个字符	$O(\log n + len)$
`r.append(str)`	在末尾追加字符串	$O(\log n + m)$
`r.clear()`	清空所有内容	$O(n)$
`r = r1 + r2`	连接两个 rope	$O(\log n + \log m)$

2. **访问与遍历**

方法	说明	时间复杂度
`r[i]`	访问第 i 个字符（无边界检查）	$O(\log n)$
`r.at(i)`	访问第 i 个字符（有边界检查）	$O(\log n)$
`r.begin()`, `r.end()`	返回正向迭代器	$O(1)$
`r.rbegin()`, `r.rend()`	返回反向迭代器	$O(1)$

方法	说明	时间复杂度
`r.c_str()`	转换为 C 风格字符串	$O(n)$

3. **容量与信息**

方法	说明	时间复杂度
`r.size()`	返回字符数量	$O(1)$
`r.length()`	返回字符数量（同 size）	$O(1)$
`r.empty()`	判断是否为空	$O(1)$
`r.max_size()`	返回最大可能大小	$O(1)$

4. **查找与比较**

方法	说明	时间复杂度
`r.find(str, pos)`	从 pos 开始查找字符串（实现依赖）	$O(n+m)$
`r.compare(pos, len, str)`	与字符串比较	$O(len)$
`==`, `!=`, `<`, `>` 等	关系运算符	$O(\min(n,m))$

5. **特殊操作**

方法	说明	时间复杂度
`r.copy(pos, len, str)`	复制到字符串 str	$O(len)$
`swap(r1, r2)`	交换两个 rope 内容	$O(1)$
`r.reserve(size)`	预留空间（实现依赖）	实现相关
`r.capacity()`	返回当前容量（实现依赖）	$O(1)$

6. **迭代器操作**

方法	说明	时间复杂度
`r.begin() + n`	前进 n 个位置	$O(\log n)$
`r.end() - n`	后退 n 个位置	$O(\log n)$
`distance(it1, it2)`	计算迭代器间距离	$O(\log n)$
`advance(it, n)`	迭代器前进 n 步	$O(\log n + \lvert n \rvert)$

适用场景

> ✓ 正确应用

1. **频繁的中间插入/删除**：如文本编辑器、IDE
2. **超大字符串处理**：如基因组序列分析
3. **版本控制系统**：需要频繁修改的文本
4. **字符串拼接/拆分频繁**的场景

✕ 错误应用

1. **空间开销**：比 `std::string` 更大，因为有额外的树结构
2. **访问延迟**：随机访问稍慢，但批量操作更快
3. **不适合**：只有随机访问需求的场景（用 `std::string` 更好）

Problem - Code