



# 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
<code>rope&lt;char&gt; r / crope r</code>	构造空 rope	$O(1)$
<code>crope r("text")</code>	使用字符串构造	$O(n)$
<code>r.push_back(c)</code>	在末尾添加字符	$O(\log n)$
<code>r.insert(pos, str)</code>	在位置 pos 插入字符串	$O(\log n + m)$
<code>r.erase(pos, len)</code>	从 pos 开始删除 len 个字符	$O(\log n)$
<code>r.replace(pos, str)</code>	从 pos 开始替换为字符串	$O(\log n + m)$
<code>r.substr(pos, len)</code>	从 pos 开始截取 len 个字符	$O(\log n + len)$
<code>r.append(str)</code>	在末尾追加字符串	$O(\log n + m)$
<code>r.clear()</code>	清空所有内容	$O(n)$
<code>r = r1 + r2</code>	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
<code>r[i]</code>	访问第 i 个字符（无边界检查）	$O(\log n)$
<code>r.at(i)</code>	访问第 i 个字符（有边界检查）	$O(\log n)$
<code>r.begin(), r.end()</code>	返回正向迭代器	$O(1)$
<code>r.rbegin(), r.rend()</code>	返回反向迭代器	$O(1)$

方法	说明	时间复杂度
<code>r.c_str()</code>	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
<code>r.size()</code>	返回字符数量	$O(1)$
<code>r.length()</code>	返回字符数量（同 size）	$O(1)$
<code>r.empty()</code>	判断是否为空	$O(1)$
<code>r.max_size()</code>	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
<code>r.find(str, pos)</code>	从 pos 开始查找字符串（实现依赖）	$O(n + m)$
<code>r.compare(pos, len, str)</code>	与字符串比较	$O(len)$
<code>==</code> , <code>!=</code> , <code>&lt;</code> , <code>&gt;</code> 等	关系运算符	$O(\min(n, m))$

### 5. 特殊操作

方法	说明	时间复杂度
<code>r.copy(pos, len, str)</code>	复制到字符串 str	$O(len)$
<code>swap(r1, r2)</code>	交换两个 rope 内容	$O(1)$
<code>r.reserve(size)</code>	预留空间（实现依赖）	实现相关
<code>r.capacity()</code>	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
<code>r.begin() + n</code>	前进 n 个位置	$O(\log n)$
<code>r.end() - n</code>	后退 n 个位置	$O(\log n)$
<code>distance(it1, it2)</code>	计算迭代器间距离	$O(\log n)$
<code>advance(it, n)</code>	迭代器前进 n 步	$O(\log n +  n )$

## 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