



PBDS

Preface 前言

pb_ds 库全称 Policy-Based Data Structures。是编译器封装库，并不是原生 `c++` 标准的内容。

在 NOI 系列活动中使用 pb_ds 库的合规性是有文件上的依据的，赛场上可以放心用。

其中封装了四种容器：平衡树，字典树，哈希表，和可并堆。常数较小。

Introduction 引子

pb_ds 的使用和其他 STL 相同，也需要引入头文件。以下是所需的头文件。

```
#include <ext/pb_ds/assoc_container.hpp> //关联容器，必加
#include <ext/pb_ds/hash_policy.hpp> //哈希表
#include <ext/pb_ds/tree_policy.hpp> //平衡树
#include <ext/pb_ds/trie_policy.hpp> //字典树
#include <ext/pb_ds/priority_queue.hpp> //优先队列
```

什么？背不下来？巧了，pb_ds 也有 万能头：

```
#include <bits/extc++.h>
```

你可能会注意到 pb_ds 中的 priority_queue 跟 C++ STL 中的 priority_queue 重名了，实际上 pb_ds 中的容器和算法都定义在以下两个命名空间里：

```
using namespace __gnu_cxx;
using namespace __gnu_pbds;
```

于是就避免了重名的问题。

为了兼顾 std 和 pbds，是时候对命名空间出手了！

```
#define endl '\n'
using std::cin;
using std::cerr;
using std::cout;
using namespace __gnu_pbds;
```

Data Structures 数据结构!

平衡树 Tree

声明

```
template<
    typename Key,           // 键类型
    typename Mapped,        // 值类型 (null_type 表示集合)
    typename Cmp_Fn = less<Key>, // 比较函数
    typename Tag = rb_tree_tag, // 树类型
    template<
        typename Const_Node_Iterator,
        typename Node_Iterator,
        typename Cmp_Fn_,
        typename Allocator_>
    class Node_Update = null_node_update // 节点更新策略
> class tree;
```

一般来说，我们使用平衡树来管理有序集合，且使用 红黑树 作为较为均衡的底层实现结构，于是得到平常在竞赛环境中常用的声明模板。

1. 实现一个整数序列（升序【第 k 个 = 第 k 大】）。

```
tree<
    int,
    null_type,less<int>,
    rb_tree_tag,
    tree_order_statistics_node_update
> tr;
```

2. 自定义类型序列（需要重载小于运算符，【第 k 个 = 第 k 小】）

```
tree<
    Type,
    null_type,
    greater<Type>,
    rb_tree_tag,
    tree_order_statistics_node_update
> tr;
```

用法

方法	说明	时间复杂度
tr.insert(x)	插入元素 x (返回 pair<iterator,bool>)	O(log n)
tr.erase(x)	删除键为 x 的元素 (返回删除个数)	O(log n)
tr.erase(it)	通过迭代器删除元素	O(log n)
tr.find(x)	查找元素，返回迭代器或 end()	O(log n)

方法	说明	时间复杂度
<code>tr.find_by_order(k)</code>	返回第 k (0-based) 个元素的迭代器；越界返回 <code>end()</code>	$O(\log n)$
<code>tr.order_of_key(x)</code>	返回严格小于 x 的元素个数	$O(\log n)$
<code>tr.lower_bound(x)</code>	返回第一个不小于 x 的迭代器	$O(\log n)$
<code>tr.upper_bound(x)</code>	返回第一个大于 x 的迭代器	$O(\log n)$
<code>tr.begin() / tr.end()</code>	正序遍历迭代器	$O(1)$
<code>tr.rbegin() / tr.rend()</code>	反向遍历迭代器	$O(1)$
<code>tr.size()</code>	返回元素数量	$O(1)$
<code>tr.empty()</code>	判断是否为空	$O(1)$
<code>tr.clear()</code>	清空容器	$O(n)$



Attention

- 平衡树为集合语义，会去重；如需保留重复元素可用 `pair<T,int>` 等技巧。
- `find_by_order` 与 `order_of_key` 依赖于使用 `tree_order_statistics_node_update` 的声明。
- 迭代器失效规则与 STL 容器类似（插入/删除可能影响迭代器）。

Problem - Code

注意和所有平衡树数据结构一样（例如 `std::map<>`），这个平衡树也会去重，若想保留多个值，可以使用 `std::pair<int,int>`，以此制造元素之间的差异。

哈希表

`gp_hash_table` 是 pbds 中的一个哈希表实现，比标准库的 `unordered_map` 快 2 – 3 倍。它属于 `__gnu_pbds` 命名空间。

基础用法

Function	描述	时间复杂度
<code>operator[]</code>	访问/插入元素	$O(1)$ 均摊
<code>find(key)</code>	查找元素	$O(1)$ 均摊
<code>erase(key)</code>	删除元素	$O(1)$ 均摊
<code>size()</code>	返回元素数量	$O(1)$
<code>empty()</code>	检查是否为空	$O(1)$
<code>clear()</code>	清空所有元素	$O(n)$

自定义哈希函数

以下段落为 AI 辅助生成，不保证正确性。

```
#include <bits/stdc++.h>
#include <ext/pb_ds/assoc_container.hpp>
using namespace std;
using namespace __gnu_pbds;

// 自定义哈希结构体
struct custom_hash {
    static uint64_t splitmix64(uint64_t x) {
        x += 0x9e3779b97f4a7c15;
        x = (x ^ (x >> 30)) * 0xbff58476d1ce4e5b9;
        x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
        return x ^ (x >> 31);
    }

    size_t operator()(uint64_t x) const {
        static const uint64_t FIXED_RANDOM = chrono::steady_clock::now()
            .time_since_epoch().count();
        return splitmix64(x + FIXED_RANDOM);
    }
};

// 使用自定义哈希
gp_hash_table<int, int, custom_hash> safe_table;

// 对于字符串键
gp_hash_table<string, int> string_table;
```

注意事项

1. **迭代器稳定性**: 插入操作可能使所有迭代器失效 (类似 `vector`)
2. **内存释放**: `clear()` 不会立即释放内存, 可用 `swap` 技巧:

```
gp_hash_table<int, int>().swap(table);
```

与 `unordered_map` 的主要区别

特性	<code>gp_hash_table</code>	<code>unordered_map</code>
实现方法	开放寻址法	链地址法
默认性能	通常更快	较慢
内存局部性	更好	较差
标准兼容	非标准	C++ 标准
哈希函数	更简单快速	更安全复杂

常用模板

```
template<typename K, typename V>
using hash_map = gp_hash_table<K, V>;
```