C++的pb_ds库在OI中的应用

大连市第二十四中学　于纪平

WC2015营员交流

2015 年 2 月 9 日

自我介绍

自我介绍

* 大连市第二十四中学　于纪平

自我介绍

* 大连市第二十四中学　于纪平
* 高三准退役Oier

自我介绍

* 大连市第二十四中学　于纪平
* 高三准退役Oier
* 171792544，saffah，saffah@126.com

选题原因

选题原因

* 困? “全国青少年信息学奥林匹克冬眠营”?

选题原因

* 困? “全国青少年信息学奥林匹克冬眠营”?
* 喜闻乐见的pb_ds库会让你眼前一亮!

选题原因

* 困? “全国青少年信息学奥林匹克冬眠营”?
* 喜闻乐见的pb_ds库会让你眼前一亮!
* OI界普及率不是很高，相关的中文资料很少

选题原因

* 困？“全国青少年信息学奥林匹克冬眠营”？
* 喜闻乐见的pb_ds库会让你眼前一亮！
* OI界普及率不是很高，相关的中文资料很少
* 只要998，骗分神器带回家！

pb_ds

pb_ds

* 平板电视?

pb_ds

* 平板电视?
* Policy-Based Data Structures

pb_ds

* 平板电视?
* Policy-Based Data Structures
* It is designed for high-performance, flexibility, semantic safety, and conformance to the corresponding containers in std and std::tr1.

pb_ds

* 平板电视?
* Policy-Based Data Structures
* It is designed for high-performance, flexibility, semantic safety, and conformance to the corresponding containers in std and std::tr1.
* 封装了很多数据结构，比如平衡树、哈希表、字典树、堆等

Why pb_ds?

Why pb_ds?

* 就像vector、set、map一样

Why pb_ds?

* 就像vector、set、map一样
* 比STL功能更多

Why pb_ds?

* 就像vector、set、map一样
* 比STL功能更多
* 按照现行NOI正式比赛规则，pb_ds库是可以使用的

Why pb_ds?

* 就像vector、set、map一样
* 比STL功能更多
* 按照现行NOI正式比赛规则，pb_ds库是可以使用的
* （即使不行也可以用来对拍）

__gnu_pbds::priority_queue

基本用法

基本用法

* 需要ext/pb_ds/priority_queue.hpp

基本用法

* 需要ext/pb_ds/priority_queue.hpp
* __gnu_pbds::priority_queue <T>

基本用法

* 需要ext/pb_ds/priority_queue.hpp
* __gnu_pbds::priority_queue <T>
* 与std::priority_queue的基本用法相同，包括size()，empty()，push(const T)，top()，pop()，clear()

模板参数

模板参数

```
template<typename Value_Type,
 typename Cmp_Fn = std::less<Value_Type>,
 typename Tag = pairing_heap_tag,
 typename Allocator = std::allocator<char> >
class priority_queue
```

模板参数

```
template<typename Value_Type,
 typename Cmp_Fn = std::less<Value_Type>,
 typename Tag = pairing_heap_tag,
 typename Allocator = std::allocator<char> >
class priority_queue
```

* Value_Type和Cmp_Fn不用说了

模板参数

```
template<typename Value_Type,
 typename Cmp_Fn = std::less<Value_Type>,
 typename Tag = pairing_heap_tag,
 typename Allocator = std::allocator<char> >
class priority_queue
```

* Value_Type和Cmp_Fn不用说了
* Tag表示所使用的堆的类型，可以是binary_heap_tag（二叉堆），binomial_heap_tag（二项堆），rc_binomial_heap_tag，pairing_heap_tag（配对堆），thin_heap_tag

模板参数

```
template<typename Value_Type,
 typename Cmp_Fn = std::less<Value_Type>,
 typename Tag = pairing_heap_tag,
 typename Allocator = std::allocator<char> >
class priority_queue
```

* Value_Type和Cmp_Fn不用说了
* Tag表示所使用的堆的类型，可以是binary_heap_tag（二叉堆），binomial_heap_tag（二项堆），rc_binomial_heap_tag，pairing_heap_tag（配对堆），thin_heap_tag
* Allocator应该不用管了

更多用法

更多用法

* 可以用begin()和end()来获取iterator从而遍历

更多用法

* 可以用begin()和end()来获取iterator从而遍历
* 可以increase_key，decrease_key，和删除单个元素

更多用法

* 可以用begin()和end()来获取iterator从而遍历
* 可以increase_key，decrease_key，和删除单个元素
* 可以合并

更多用法

更多用法

* point_iterator push(const_reference)

更多用法

* point_iterator push(const_reference)
* void modify(point_iterator, const_reference)

更多用法

* point_iterator push(const_reference)
* void modify(point_iterator, const_reference)
* void erase(point_iterator)

更多用法

* point_iterator push(const_reference)
* void modify(point_iterator, const_reference)
* void erase(point_iterator)

```
priority_queue<int> p;
priority_queue<int>::point_iterator it = p.push(0);
p.push(1); p.push(2);
p.modify(it, 3);
assert(p.top() == 3);
p.erase(it);
assert(p.top() == 2);
```

更多用法

更多用法

* void join(priority_queue &other)

更多用法

* void join(priority_queue &other)
* 把other合并到*this，然后other会被清空

例题

例题

* 需要例题吗?

例题

* 需要例题吗?
* 白送一个可并堆，想怎么用怎么用！

时间复杂度分析

时间复杂度分析

* 共有五种操作：push、pop、modify、erase、join

时间复杂度分析

* 共有五种操作：push、pop、modify、erase、join
* pairing_heap_tag：push和join为$O(1)$，其余为均摊$\Theta(\log n)$

时间复杂度分析

* 共有五种操作：push、pop、modify、erase、join
* pairing_heap_tag：push和join为$O(1)$，其余为均摊$\Theta(\log n)$
* binary_heap_tag：只支持push和pop，均为均摊$\Theta(\log n)$

时间复杂度分析

* 共有五种操作：push、pop、modify、erase、join
* pairing_heap_tag：push和join为$O(1)$，其余为均摊$\Theta(\log n)$
* binary_heap_tag：只支持push和pop，均为均摊$\Theta(\log n)$
* binomial_heap_tag：push为均摊$O(1)$，其余为$\Theta(\log n)$

时间复杂度分析

* 共有五种操作：push、pop、modify、erase、join
* pairing_heap_tag：push和join为$O(1)$，其余为均摊$\Theta(\log n)$
* binary_heap_tag：只支持push和pop，均为均摊$\Theta(\log n)$
* binomial_heap_tag：push为均摊$O(1)$，其余为$\Theta(\log n)$
* rc_binomial_heap_tag：push为$O(1)$，其余为$\Theta(\log n)$

时间复杂度分析

* 共有五种操作：push、pop、modify、erase、join
* pairing_heap_tag：push和join为$O(1)$，其余为均摊$\Theta(\log n)$
* binary_heap_tag：只支持push和pop，均为均摊$\Theta(\log n)$
* binomial_heap_tag：push为均摊$O(1)$，其余为$\Theta(\log n)$
* rc_binomial_heap_tag：push为$O(1)$，其余为$\Theta(\log n)$
* thin_heap_tag：push为$O(1)$，不支持join，其余为$\Theta(\log n)$；但是如果只有increase_key，那么modify为均摊$O(1)$

时间复杂度分析

* 共有五种操作：push、pop、modify、erase、join
* pairing_heap_tag：push和join为$O(1)$，其余为均摊$\Theta(\log n)$
* binary_heap_tag：只支持push和pop，均为均摊$\Theta(\log n)$
* binomial_heap_tag：push为均摊$O(1)$，其余为$\Theta(\log n)$
* rc_binomial_heap_tag：push为$O(1)$，其余为$\Theta(\log n)$
* thin_heap_tag：push为$O(1)$，不支持join，其余为$\Theta(\log n)$；但是如果只有increase_key，那么modify为均摊$O(1)$
* “不支持”不是不能用，而是用起来很慢

时间复杂度分析

时间复杂度分析

* 通常需要合并的时候就用pairing_heap_tag

时间复杂度分析

* 通常需要合并的时候就用pairing_heap_tag
* 做Dijkstra的时候就用thin_heap_tag

时间复杂度分析

* 通常需要合并的时候就用pairing_heap_tag
* 做Dijkstra的时候就用thin_heap_tag
* 真的是这样吗?

时间复杂度分析

* 通常需要合并的时候就用pairing_heap_tag
* 做Dijkstra的时候就用thin_heap_tag
* 真的是这样吗?
* 实践是检验真理的唯一标准!

性能测试

性能测试

* 参与测试的有：std::priority_queue，binary_heap_tag，binomial_heap_tag，pairing_heap_tag，rc_binomial_heap_tag，thin_heap_tag，手写线段树

性能测试

* 参与测试的有：std::priority_queue，binary_heap_tag，binomial_heap_tag，pairing_heap_tag，rc_binomial_heap_tag，thin_heap_tag，手写线段树
* 评测机：Windows，i5-3210M 2.50GHz，gcc version 4.4.0 (GCC)

测试一：堆排序

测试一：堆排序

* 随机生成10^7个$[0, 2^{30})$的整数，插入到数据结构中，然后pop出来

测试一：堆排序

* 随机生成10^7个$[0, 2^{30})$的整数，插入到数据结构中，然后pop出来

数据结构	不开O2的用时	开O2的用时
std::priority_queue	16378 ms	2797 ms
binary_heap_tag	5454 ms	2720 ms
pairing_heap_tag	21421 ms	17362 ms
thin_heap_tag	39290 ms	28672 ms
手写线段树	8692 ms	4916 ms
(*)push_back+sort+pop_back	4563 ms	938 ms

测试二：堆的合并

测试二：堆的合并

* 随机生成4194304个$[0, 2^{30})$的整数，为每个数建一个堆，然后像归并排序一样两两合并

测试二：堆的合并

* 随机生成4194304个$[0, 2^{30})$的整数，为每个数建一个堆，然后像归并排序一样两两合并
* 注意binary_heap_tag的join不是启发式合并，时间与较大堆的大小成正比；而thin_heap_tag是启发式合并

测试二：堆的合并

* 随机生成4194304个$[0, 2^{30})$的整数，为每个数建一个堆，然后像归并排序一样两两合并
* 注意binary_heap_tag的join不是启发式合并，时间与较大堆的大小成正比；而thin_heap_tag是启发式合并

数据结构	不开O2的用时	开O2的用时
std::priority_queue+手写启发式合并	38746 ms	4642 ms
(*)binary_heap_tag	5605 ms	3127 ms
binary_heap_tag+手写启发式合并	11206 ms	3955 ms
pairing_heap_tag	6954 ms	5328 ms
thin_heap_tag	14514 ms	9588 ms

测试三：堆优化Dijkstra算法

测试三：堆优化Dijkstra算法

* 构造一个10^6点，2×10^7边的有向图，使得堆优化Dijkstra算法的时间复杂度达到最坏情况

测试三：堆优化Dijkstra算法

* 构造一个10^6点，2×10^7边的有向图，使得堆优化Dijkstra算法的时间复杂度达到最坏情况
* 注意binary_heap_tag不支持这种操作

测试三：堆优化Dijkstra算法

* 构造一个10^6点，2×10^7边的有向图，使得堆优化Dijkstra算法的时间复杂度达到最坏情况
* 注意binary_heap_tag不支持这种操作

数据结构	不开O2的用时	开O2的用时
binomial_heap_tag	26807 ms	17018 ms
pairing_heap_tag	7684 ms	6230 ms
rc_binomial_heap_tag	32004 ms	18745 ms
thin_heap_tag	9388 ms	6483 ms
手写线段树	9595 ms	5080 ms

测试三：堆优化Dijkstra算法

测试三：堆优化Dijkstra算法

* 构造一个n点，m边的无向图，使得堆优化Dijkstra算法的时间复杂度达到较坏情况

测试三：堆优化Dijkstra算法

* 构造一个n点，m边的无向图，使得堆优化Dijkstra算法的时间复杂度达到较坏情况
* 十分感谢lydrainbowcat提供测试数据和手写数据结构

测试三：堆优化Dijkstra算法

* 构造一个n点，m边的无向图，使得堆优化Dijkstra算法的时间复杂度达到较坏情况
* 十分感谢lydrainbowcat提供测试数据和手写数据结构
* （只计Dijkstra算法运行的时间，不计生成数据等的时间）

不开O2

不开O2

n	m	pairing	thin	Fib堆	二叉堆	线段树
100k	10M	390 ms	578 ms	703 ms	1088 ms	422 ms
1M	10M	2063 ms	3516 ms	4209 ms	9236 ms	2594 ms
1M	20M	4298 ms	6595 ms	6859 ms	18648 ms	4745 ms
100k	25M	3375 ms	4805 ms	3735 ms	5655 ms	2501 ms
1M	30M	6579 ms	9609 ms	9393 ms	27960 ms	6816 ms
50k	50M	484 ms	563 ms	1235 ms	6392 ms	453 ms
100k	50M	7061 ms	9550 ms	8051 ms	11301 ms	4580 ms
1M	50M	11088 ms	15559 ms	14707 ms	47108 ms	10848 ms

开O2

开O2

n	m	pairing	thin	Fib堆	二叉堆	线段树
100k	10M	281 ms	360 ms	438 ms	625 ms	219 ms
1M	10M	1485 ms	2250 ms	2836 ms	7968 ms	1444 ms
1M	20M	3204 ms	4485 ms	4987 ms	16629 ms	2711 ms
100k	25M	2141 ms	2798 ms	3206 ms	2309 ms	1262 ms
1M	30M	4908 ms	6642 ms	6795 ms	25128 ms	3964 ms
50k	50M	375 ms	407 ms	680 ms	2217 ms	280 ms
100k	50M	4245 ms	5792 ms	4150 ms	4105 ms	2312 ms
1M	50M	8455 ms	11541 ms	10927 ms	42101 ms	6217 ms

大致结论

大致结论

* 对于优化Dijkstra算法，pairing_heap_tag严格快于thin_heap_tag，并且无论是否开O2，速度都大致与手写数据结构相当

大致结论

* 对于优化Dijkstra算法，pairing_heap_tag严格快于thin_heap_tag，并且无论是否开O2，速度都大致与手写数据结构相当
* 线段树大法好

大致结论

大致结论

* binary_heap_tag在绝大多数情况下优于std::priority_queue

大致结论

* binary_heap_tag在绝大多数情况下优于std::priority_queue
* pairing_heap_tag在绝大多数情况下优于binomial_heap_tag和rc_binomial_heap_tag

大致结论

* binary_heap_tag在绝大多数情况下优于std::priority_queue
* pairing_heap_tag在绝大多数情况下优于binomial_heap_tag和rc_binomial_heap_tag
* 在只有push，pop和join操作时，binary_heap_tag速度较快

大致结论

* binary_heap_tag在绝大多数情况下优于std::priority_queue
* pairing_heap_tag在绝大多数情况下优于binomial_heap_tag和rc_binomial_heap_tag
* 在只有push，pop和join操作时，binary_heap_tag速度较快
* 在有modify操作时，可以考虑采用pairing_heap_tag，thin_heap_tag或手写数据结构

__gnu_pbds::tree

基本用法

基本用法

* 需要ext/pb_ds/assoc_container.hpp和ext/pb_ds/tree_policy.hpp

基本用法

* 需要ext/pb_ds/assoc_container.hpp和ext/pb_ds/tree_policy.hpp
* __gnu_pbds::tree <Key, T>

基本用法

* 需要ext/pb_ds/assoc_container.hpp和ext/pb_ds/tree_policy.hpp
* __gnu_pbds::tree <Key, T>
* 与std::map的基本用法相同，包括begin(), end(), size(), empty(), clear(), find(const Key), lower_bound(const Key), upper_bound(const Key), erase(iterator), erase(const Key), insert(const pair<Key, T>), operator[](const Key)

基本用法

基本用法

* 不想用map而是set? 只需要把第二个模板参数改成null_type就可以了!

基本用法

* 不想用map而是set？只需要把第二个模板参数改成null_type就可以了！
* （注意，在比较旧的版本，例如4.4.0，需要使用null_mapped_type）

基本用法

* 不想用map而是set？只需要把第二个模板参数改成null_type就可以了！
* （注意，在比较旧的版本，例如4.4.0，需要使用null_mapped_type）
* 此时迭代器指向的类型也会从pair变为Key，也就是和set几乎没有区别

模板参数

模板参数

```
template<
    typename Key, typename Mapped,
    typename Cmp_Fn = std::less<Key>,
    typename Tag = rb_tree_tag,
    template<
        typename Const_Node_Iterator,
        typename Node_Iterator,
        typename Cmp_Fn_, typename Allocator_>
    class Node_Update = null_tree_node_update,
    typename Allocator = std::allocator<char> >
class tree;
```

模板参数

模板参数

```
typename Key, typename Mapped,
typename Cmp_Fn = std::less<Key>,
typename Tag = rb_tree_tag,
```

模板参数

```
typename Key, typename Mapped,
typename Cmp_Fn = std::less<Key>,
typename Tag = rb_tree_tag,
```

* 前三个参数Key，Mapped，Cmp_Fn都不用说了

模板参数

```
typename Key, typename Mapped,
typename Cmp_Fn = std::less<Key>,
typename Tag = rb_tree_tag,
```

* 前三个参数Key，Mapped，Cmp_Fn都不用说了
* Tag是指tree的类型，可以是rb_tree_tag，splay_tree_tag，ov_tree_tag之一

开心一刻

开心一刻

* 前段时间，在noip贴吧中出现了很多追捧Trajan的spaly的朋友，也有坚守splay不动摇的朋友

开心一刻

* 前段时间，在noip贴吧中出现了很多追捧Trajan的spaly的朋友，也有坚守splay不动摇的朋友
* 不要问我资瓷什么，我要问pb_ds资瓷什么！

开心一刻

* 前段时间，在noip贴吧中出现了很多追捧Trajan的spaly的朋友，也有坚守splay不动摇的朋友
* 不要问我资瓷什么，我要问pb_ds资瓷什么！
* 让我们看看ext/pb_ds/detail/splay_tree/splay_tree.hpp的132行！

开心一刻

* 前段时间，在noip贴吧中出现了很多追捧Trajan的spaly的朋友，也有坚守splay不动摇的朋友
* 不要问我资瓷什么，我要问pb_ds资瓷什么！
* 让我们看看ext/pb_ds/detail/splay_tree/splay_tree.hpp的132行！

```
// $p14y 7r33 7481.
template<typename Key, typename Mapped, typename Cmp_Fn,
    typename Node_And_It_Traits, typename Allocator>
class PB_DS_CLASS_NAME : public PB_DS_BASE_C_DEC
```

开心一刻

* 前段时间，在noip贴吧中出现了很多追捧Trajan的spaly的朋友，也有坚守splay不动摇的朋友
* 不要问我资瓷什么，我要问pb_ds资瓷什么！
* 让我们看看ext/pb_ds/detail/splay_tree/splay_tree.hpp的132行！

```
// $p14y 7r33 7481.
template<typename Key, typename Mapped, typename Cmp_Fn,
    typename Node_And_It_Traits, typename Allocator>
class PB_DS_CLASS_NAME : public PB_DS_BASE_C_DEC
```

* pb_ds使用的表面是splay tree，但实际上是一种叫做$p14y 7r33的高深数据结构！

$p14y 7r33 7481.

$p14y 7r33 7481.

* 看上去应该是leetspeak

$p14y 7r33 7481.

* 看上去应该是leetspeak
* splay tree tabl? 好像说不通?

$p14y 7r33 7481.

* 看上去应该是leetspeak
* splay tree tabl? 好像说不通?
* 如果现场有懂这个的欢迎发表自己的见解

模板参数

模板参数

```
template<
    typename Const_Node_Iterator,
    typename Node_Iterator,
    typename Cmp_Fn_, typename Allocator_>
class Node_Update = null_tree_node_update,
```

模板参数

```
template<
    typename Const_Node_Iterator,
    typename Node_Iterator,
    typename Cmp_Fn_, typename Allocator_>
class Node_Update = null_tree_node_update,
```

* 重点是这个Node_Update，不看它这个class的模板参数，只看默认值是个空

模板参数

```
template<
    typename Const_Node_Iterator,
    typename Node_Iterator,
    typename Cmp_Fn_, typename Allocator_>
class Node_Update = null_tree_node_update,
```

* 重点是这个Node_Update，不看它这个class的模板参数，只看默认值是个空
* pb_ds库中也提供了一个非空的选项，比如自带的tree_order_statistics_node_update

模板参数

```
template<
    typename Const_Node_Iterator,
    typename Node_Iterator,
    typename Cmp_Fn_, typename Allocator_>
class Node_Update = null_tree_node_update,
```

* 重点是这个Node_Update，不看它这个class的模板参数，只看默认值是个空
* pb_ds库中也提供了一个非空的选项，比如自带的tree_order_statistics_node_update
* 只要将Node_Update设定为tree_order_statistics_node_update，这个tree就会获得两个函数find_by_order和order_of_key

更多用法

更多用法

* iterator find_by_order(size_type order)

更多用法

* iterator find_by_order(size_type order)
* 找第order+1小的元素的迭代器，如果order太大会返回end()

更多用法

* iterator find_by_order(size_type order)
* 找第order+1小的元素的迭代器，如果order太大会返回end()
* size_type order_of_key(const_key_reference r_key)

更多用法

* iterator find_by_order(size_type order)
* 找第order+1小的元素的迭代器，如果order太大会返回end()
* size_type order_of_key(const_key_reference r_key)
* 询问这个tree中有多少个比r_key小的元素

更多用法

更多用法

* void join(tree &other)

更多用法

* void join(tree &other)
* 把other的所有元素移动到*this中（要求原来other和*this的值域不能相交，否则会抛出异常）

更多用法

* void join(tree &other)
* 把other的所有元素移动到*this中（要求原来other和*this的值域不能相交，否则会抛出异常）
* void split(const_key_reference r_key, tree &other)

更多用法

* void join(tree &other)
* 把other的所有元素移动到*this中（要求原来other和*this的值域不能相交，否则会抛出异常）
* void split(const_key_reference r_key, tree &other)
* 清空other，然后把*this当中所有大于r_key的元素移至other

例题

例题

* 需要例题吗?

例题

* 需要例题吗?
* 白送一个支持查询rank和k小值的set/map，想怎么用怎么用!

自定义Node_Update

自定义Node_Update

* 自带的tree_order_statistics_node_update统计的是子树size

自定义Node_Update

* 自带的tree_order_statistics_node_update统计的是子树size
* 稍加修改就可以统计容易合并的任意信息

自定义Node_Update

* 自带的tree_order_statistics_node_update统计的是子树size
* 稍加修改就可以统计容易合并的任意信息
* 比如要做一个能查询子段和的map <int, int>，就需要记录子树的mapped_value之和

自定义Node_Update

* 自带的tree_order_statistics_node_update统计的是子树size
* 稍加修改就可以统计容易合并的任意信息
* 比如要做一个能查询子段和的map <int, int>，就需要记录子树的mapped_value之和
* 就让我们欢脱地自定义Node_Update吧！

自定义Node_Update

自定义Node_Update

```
template<class Node_CItr, class Node_Itr,
    class Cmp_Fn, class _Alloc>
struct my_node_update{
    virtual Node_CItr node_begin() const = 0;
    virtual Node_CItr node_end() const = 0;
    typedef int metadata_type;
```

自定义Node_Update

```
template<class Node_CItr, class Node_Itr,
    class Cmp_Fn, class _Alloc>
struct my_node_update{
    virtual Node_CItr node_begin() const = 0;
    virtual Node_CItr node_end() const = 0;
    typedef int metadata_type;
```

* 前面的都是固定格式

自定义Node_Update

```
template<class Node_CItr, class Node_Itr,
    class Cmp_Fn, class _Alloc>
struct my_node_update{
    virtual Node_CItr node_begin() const = 0;
    virtual Node_CItr node_end() const = 0;
    typedef int metadata_type;
```

* 前面的都是固定格式
* metadata_type是指节点上记录的额外信息的类型，这里取int

自定义Node_Update

自定义Node_Update

```
inline void operator()(Node_Itr it, Node_CItr end_it){
    Node_Itr l = it.get_l_child(), r = it.get_r_child();
    int left = 0, right = 0;
    if(l != end_it) left = l.get_metadata();
    if(r != end_it) right = r.get_metadata();
    const_cast<metadata_type &>(it.get_metadata())
        = left + right + (*it)->second;
}
```

自定义Node_Update

```
inline void operator()(Node_Itr it, Node_CItr end_it){
    Node_Itr l = it.get_l_child(), r = it.get_r_child();
    int left = 0, right = 0;
    if(l != end_it) left = l.get_metadata();
    if(r != end_it) right = r.get_metadata();
    const_cast<metadata_type &>(it.get_metadata())
        = left + right + (*it)->second;
}
```

* operator()的功能是是将节点it的信息更新为其左右孩子的信息之和，传入的end_it表示空节点

自定义Node_Update

```
inline void operator()(Node_Itr it, Node_CItr end_it){
    Node_Itr l = it.get_l_child(), r = it.get_r_child();
    int left = 0, right = 0;
    if(l != end_it) left = l.get_metadata();
    if(r != end_it) right = r.get_metadata();
    const_cast<metadata_type &>(it.get_metadata())
        = left + right + (*it)->second;
}
```

* operator()的功能是是将节点it的信息更新为其左右孩子的信息之和，传入的end_it表示空节点
* it是Node_Itr，用星号进行取值后变成了iterator，再取->second之后就是这个节点的mapped_value

自定义Node_Update

自定义Node_Update

```
inline int prefix_sum(int x){
    int ans = 0;
    Node_CItr it = node_begin();
    while(it != node_end()){
        Node_CItr l = it.get_l_child(), r = it.get_r_child();
        if(Cmp_Fn()(x, (*it)->first)) it = l;
        else{
            ans += (*it)->second;
            if(l != node_end()) ans += l.get_metadata();
            it = r;
        }
    }
    return ans;
}
```

自定义Node_Update

自定义Node_Update

```
inline int interval_sum(int l, int r){
    return prefix_sum(r) - prefix_sum(l - 1);
}
```

自定义Node_Update

```
inline int interval_sum(int l, int r){
    return prefix_sum(r) - prefix_sum(l - 1);
}
```

* 然后就可以大功告成啦!

自定义Node_Update

```
inline int interval_sum(int l, int r){
    return prefix_sum(r) - prefix_sum(l - 1);
}
```

* 然后就可以大功告成啦!
* 只需记住，对Node_Itr可以做的事情有：用get_l_child，get_r_child获取左右孩子，用两个星号获取节点信息，用get_metadata获取节点额外信息

自定义Node_Update

```
inline int interval_sum(int l, int r){
    return prefix_sum(r) - prefix_sum(l - 1);
}
```

* 然后就可以大功告成啦!
* 只需记住，对Node_Itr可以做的事情有：用get_l_child，get_r_child获取左右孩子，用两个星号获取节点信息，用get_metadata获取节点额外信息
* 一开始是模板参数Node_CItr，Node_Itr，Cmp_Fn和_Alloc（这个貌似没有用），然后定义两个虚函数，再定义()操作符用于更新某个节点的信息，然后就可以想做什么做什么了!

自定义Node_Update

自定义Node_Update

```
tree<int, int, std::less<int>, rb_tree_tag, my_node_update> T;
T[2] = 100; T[3] = 1000; T[4] = 10000;
printf("%d\n", T.interval_sum(3, 4));
printf("%d\n", T.prefix_sum(3));
```

自定义Node_Update

```
tree<int, int, std::less<int>, rb_tree_tag, my_node_update> T;
T[2] = 100; T[3] = 1000; T[4] = 10000;
printf("%d\n", T.interval_sum(3, 4));
printf("%d\n", T.prefix_sum(3));
```

* 用起来还是很方便的嘛!

自定义Node_Update

自定义Node_Update

* 这样就可以得到一个功能自定义的tree了

自定义Node_Update

* 这样就可以得到一个功能自定义的tree了
* 而且附送之前的所有功能，包括insert，find，erase，甚至是join和split都是支持的

自定义Node_Update

* 这样就可以得到一个功能自定义的tree了
* 而且附送之前的所有功能，包括insert，find，erase，甚至是join和split都是支持的
* 代码短，适合把平衡树写得很长的人和~~不会~~一时忘了怎么写平衡树的人使用

自定义Node_Update

* 这样就可以得到一个功能自定义的tree了
* 而且附送之前的所有功能，包括insert，find，erase，甚至是join和split都是支持的
* 代码短，适合把平衡树写得很长的人和~~不会~~一时忘了怎么写平衡树的人使用
* 不足之处：不支持一些低级操作，例如交换左右子树

自定义Node_Update

* 这样就可以得到一个功能自定义的tree了
* 而且附送之前的所有功能，包括insert，find，erase，甚至是join和split都是支持的
* 代码短，适合把平衡树写得很长的人和~~不会~~一时忘了怎么写平衡树的人使用
* 不足之处：不支持一些低级操作，例如交换左右子树
* 如果支持打标记，需要很多冗余的代码，也就失去了用这个库的意义

性能测试

性能测试

* 时间复杂度就不分析了，直接进行测试

性能测试

* 时间复杂度就不分析了，直接进行测试
* 参与测试的是各种tree_tag和set/map

测试一：平衡树排序

测试一：平衡树排序

* 随机生成10^7个$[0, 2^{30})$的整数，插入到数据结构中，然后遍历一遍

测试一：平衡树排序

* 随机生成10^7个$[0, 2^{30})$的整数，插入到数据结构中，然后遍历一遍
* ov_tree_tag的时间复杂度是平方级的，光荣TLE

测试一：平衡树排序

* 随机生成10^7个$[0, 2^{30})$的整数，插入到数据结构中，然后遍历一遍
* ov_tree_tag的时间复杂度是平方级的，光荣TLE

数据结构	不开O2的用时	开O2的用时
std::set	16154 ms	11033 ms
rb_tree_tag	13336 ms	10909 ms
splay_tree_tag	23425 ms	15051 ms
手写splay	25157 ms	15811 ms
WJMZBMR指针版treap	20906 ms	16627 ms
(*)手写裸BST	11783 ms	9398 ms
(*)push_back+sort	4574 ms	949 ms

测试二：字典

测试二：字典

* 随机生成10^7个$[0, 2^{30})$的整数当做key，再随机生成相同数量的整数作为mapped_value，插入到数据结构中，然后随机进行10^7次询问

测试二：字典

* 随机生成10^7个$[0, 2^{30})$的整数当做key，再随机生成相同数量的整数作为mapped_value，插入到数据结构中，然后随机进行10^7次询问

数据结构	不开O2的用时	开O2的用时
std::map	29160 ms	20598 ms
rb_tree_tag	24661 ms	20644 ms
splay_tree_tag	48712 ms	32100 ms
(*)push_back+sort+lower_bound	15236 ms	5267 ms

测试二：字典

* 随机生成10^7个$[0, 2^{30})$的整数当做key，再随机生成相同数量的整数作为mapped_value，插入到数据结构中，然后随机进行10^7次询问

数据结构	不开O2的用时	开O2的用时
std::map	29160 ms	20598 ms
rb_tree_tag	24661 ms	20644 ms
splay_tree_tag	48712 ms	32100 ms
(*)push_back+sort+lower_bound	15236 ms	5267 ms
__gnu_pbds::cc_hash_table	5705 ms	3750 ms
__gnu_pbds::gp_hash_table	5173 ms	1407 ms

hash_table

hash_table

* 需要ext/pb_ds/assoc_container.hpp和ext/pb_ds/hash_policy.hpp

hash_table

* 需要ext/pb_ds/assoc_container.hpp和ext/pb_ds/hash_policy.hpp
* __gnu_pbds::cc_hash_table <Key, Mapped>

hash_table

* 需要ext/pb_ds/assoc_container.hpp和ext/pb_ds/hash_policy.hpp
* __gnu_pbds::cc_hash_table <Key, Mapped>
* __gnu_pbds::gp_hash_table <Key, Mapped>

hash_table

* 需要ext/pb_ds/assoc_container.hpp和ext/pb_ds/hash_policy.hpp
* __gnu_pbds::cc_hash_table <Key, Mapped>
* __gnu_pbds::gp_hash_table <Key, Mapped>
* 应该足够了，支持find和operator[]

总结

总结

* priority_queue，与STL相比支持了modify，erase和join

总结

* priority_queue，与STL相比支持了modify，erase和join
* tree，相当于STL的set/map，还支持split和join；运用tree_order_statistics_node_update还支持查询rank和k小值；更可以自定义Node_Update来维护更多信息

总结

* priority_queue，与STL相比支持了modify，erase和join
* tree，相当于STL的set/map，还支持split和join；运用tree_order_statistics_node_update还支持查询rank和k小值；更可以自定义Node_Update来维护更多信息
* （目前比赛环境的）STL没有的两种hash_table

总结

* priority_queue，与STL相比支持了modify，erase和join
* tree，相当于STL的set/map，还支持split和join；运用tree_order_statistics_node_update还支持查询rank和k小值；更可以自定义Node_Update来维护更多信息
* （目前比赛环境的）STL没有的两种hash_table
* 无脑用pb_ds库代替std::set/map/priority_queue不会使程序变得更慢

致谢

致谢

* 感谢CCF举办的冬令营及营员交流活动

致谢

* 感谢CCF举办的冬令营及营员交流活动
* 感谢我的母校大连市第二十四中学，以及教练员张新钢老师

致谢

* 感谢CCF举办的冬令营及营员交流活动
* 感谢我的母校大连市第二十四中学，以及教练员张新钢老师
* 感谢羊瑞同学让我知道了pb_ds这个库，以及adamant在Codeforces博客的相关介绍

致谢

谢谢观赏