Standard Code Library(ExSTL Part)

F0RE1GNERS

Jiangxi Normal University HeartFireY

September 1202

Contents

ExtC++(PBDS) 食用方法

Part1. 引入

pb_ds 库全称 Policy-Based Data Structures。

pb_ds 库封装了很多数据结构，比如哈希（Hash）表，平衡二叉树，字典树（Trie 树），堆（优先队列）等。

就像 `vector`、`set`、`map` 一样，其组件均符合 STL 的相关接口规范。部分（如优先队列）包含 STL 内对应组件的所有功能，但比 STL 功能更多。

pb_ds 只在使用 libstdc++ 为标准库的编译器下可以用。

引入方法：

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>    // 引入平衡树
#include <ext/pb_ds/hash_policy.hpp>    // 引入 hash
#include <ext/pb_ds/trie_policy.hpp>    // 引入 trie
#include <ext/pb_ds/priority_queue.hpp> // 引入 priority_queue
using namespace __gnu_pbds;
```

更为简洁的引入方式：

```
#include <bits/extc++.h>    //直接全部淦进来
using namespace __gnu_pbds;
```

Part2. 用法

(1). 平衡树：Tree

要求引入头文件：

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
```

构造方式：

```
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;
```

模板形参：

- `Key`: 储存的元素类型，如果想要存储多个相同的 `Key` 元素，则需要使用类似于 `std::pair` 和 `struct` 的方法，并配合使用 `lower_bound` 和 `upper_bound` 成员函数进行查找
- `Mapped`: 映射规则（Mapped-Policy）类型，如果要指示关联容器是 **集合**，类似于存储元素在 `std::set` 中，此处填入 `null_type`，低版本 g++ 此处为 `null_mapped_type`；如果要指示关联容器是 **带值的集合**，类似于存储元素在 `std::map` 中，此处填入类似于 `std::map<Key, Value>` 的 `Value` 类型
- `Cmp_Fn`: 关键字比较函子，例如 `std::less<Key>`
- `Tag`: 选择使用何种底层数据结构类型，默认是 `rb_tree_tag`。`__gnu_pbds` 提供不同的三种平衡树，分别是：
  - `rb_tree_tag`：红黑树，[**一般使用这个**]，后两者的性能一般不如红黑树，容易被卡
  - `splay_tree_tag`：splay 树
  - `ov_tree_tag`：有序向量树，只是一个由 `vector` 实现的有序结构，类似于排序的 `vector` 来实现平衡树，性能取决于数据想不想卡你
- `Node_Update`：用于更新节点的策略，默认使用 `null_node_update`，若要使用 `order_of_key` 和 `find_by_order` 方法，需要使用 `tree_order_statistics_node_update`(该方法是在统计子树的 $size$)

- `Allocator`：空间分配器类型

成员函数：

- `insert(x)`：向树中插入一个元素 x，返回 `std::pair<point_iterator, bool>`。
- `erase(x)`：从树中删除一个元素/迭代器 x，返回一个 `bool` 表明是否删除成功。
- `order_of_key(x)`：返回 x 以 `Cmp_Fn` 比较的排名。
- `find_by_order(x)`：返回 `Cmp_Fn` 比较的排名所对应元素的迭代器。
- `lower_bound(x)`：以 `Cmp_Fn` 比较做 `lower_bound`，返回迭代器。
- `upper_bound(x)`：以 `Cmp_Fn` 比较做 `upper_bound`，返回迭代器。
- `join(x)`：将 x 树并入当前树，前提是两棵树的类型一样，x 树被删除。
- `split(x,b)`：以 `Cmp_Fn` 比较，小于等于 x 的属于当前树，其余的属于 b 树。
- `empty()`：返回是否为空。
- `size()`：返回大小。

自义定 Node_update

```cpp
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; // metadata type: 是指节点上记录的额外信息的类型
    // operator() 的功能是将节点 it 的信息更新为其左右儿子的信息之和，传入的 end_it 表示空节点
    // it 是 Node_Iter，用星号进行取值后变为 iterator，-> second 即为该节点的 mapped_value
    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 + 1;
    }
    inline int order_of_key(pair<int, int> x) {
        int ans = 0;
        Node_CItr it = node_begin();
        while(it != node_end()) {
            Node_CItr l = it.get_l_child();
            Node_CItr r = it.get_r_child();
            if(Cmp_Fn()(x, **it)) it = l;
            else {
                ans++;
                if(l != node_end()) ans += l.get_metadata();
                it = r;
            }
        }
        return ans;
    }
};

tree<pair<int, int>, null_type, less<pair<int, int>>, rb_tree_tag, my_node_update> tr;
```

(2). 字典树 Trie

要求引入头文件

```cpp
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/trie_policy.hpp>
using namespace __gnu_pbds;
```

构造方式以及使用方法

```cpp
typedef trie<string,null_type,trie_string_access_traits<>,pat_trie_tag,trie_prefix_search_node_update> tr;
//第一个参数必须为字符串类型，tag 也有别的 tag，但 pat 最快，与 tree 相同，node_update 支持自定义
tr.insert(s); //插入 s
tr.erase(s); //删除 s
tr.join(b); //将 b 并入 tr
pair//pair 的使用如下：
```

```
pair<tr::iterator,tr::iterator> range=base.prefix_range(x);
for(tr::iterator it=range.first;it!=range.second;it++) cout<<*it<<' '<<endl;
//pair 中第一个是起始迭代器，第二个是终止迭代器，遍历过去就可以找到所有字符串了。
```

(3). 哈希表 HashTable

要求引入头文件

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/hash_policy.hpp>    // 引入 hash
using namespace __gnu_pbds;
```

使用方法

```
cc_hash_table<int, bool> h;  // 拉链法
gp_hash_table<int, bool> h;  // 探测法（推荐）
```

其余方法同 `std::map`，但是注意，该数据结构的总复杂度是 $O(N)$。

(4). 堆 Priority_queue

附：官方文档地址——复杂度及常数测试

```
#include <ext/pb_ds/priority_queue.hpp>
using namespace __gnu_pbds;
__gnu_pbds ::priority_queue<T, Compare, Tag, Allocator>
```

模板形参

- T: 储存的元素类型
- `Compare`: 提供严格的弱序比较类型
- `Tag`: 是 `__gnu_pbds` 提供的不同的五种堆，Tag 参数默认是 `pairing_heap_tag` 五种分别是：
- `pairing_heap_tag`：配对堆官方文档认为在非原生元素（如自定义结构体/`std :: string`/`pair`）中，配对堆表现最好
- `binary_heap_tag`：二叉堆官方文档认为在原生元素中二叉堆表现最好，不过我测试的表现并没有那么好
- `binomial_heap_tag`：二项堆二项堆在合并操作的表现要优于二叉堆，但是其取堆顶元素操作的复杂度比二叉堆高
- `rc_binomial_heap_tag`：冗余计数二项堆
- `thin_heap_tag`：除了合并的复杂度都和 Fibonacci 堆一样的一个 tag
- `Allocator`：空间配置器，由于 OI 中很少出现，故这里不做讲解

由于本篇文章只是提供给学习算法竞赛的同学们，故对于后四个 tag 只会简单的介绍复杂度，第一个会介绍成员函数和使用方法。

经作者本机 Core i5 @3.1 GHz On macOS 测试堆的基础操作，结合 GNU 官方的复杂度测试，Dijkstra 测试，都表明：至少对于 OIer 来讲，除了配对堆的其他四个 tag 都是鸡肋，要么没用，要么常数大到不如 `std` 的，且有可能造成 MLE，故这里只推荐用默认的配对堆。同样，配对堆也优于 `algorithm` 库中的 `make_heap()`。

构造方式

要注明命名空间因为和 `std` 的类名称重复。

```
__gnu_pbds ::priority_queue<int> __gnu_pbds::priority_queue<int, greater<int> >
__gnu_pbds ::priority_queue<int, greater<int>, pairing_heap_tag>
__gnu_pbds ::priority_queue<int>::point_iterator id; // 点类型迭代器
// 在 modify 和 push 的时候都会返回一个 point_iterator，下文会详细的讲使用方法
id = q.push(1);
```

成员函数

- `push()`: 向堆中压入一个元素，返回该元素位置的迭代器。
- `pop()`: 将堆顶元素弹出。
- `top()`: 返回堆顶元素。
- `size()` 返回元素个数。
- `empty()` 返回是否非空。
- `modify(point_iterator, const key)`: 把迭代器位置的 `key` 修改为传入的 `key`，并对底层储存结构进行排序。

- `erase(point_iterator)`: 把迭代器位置的键值从堆中擦除。
- `join(__gnu_pbds :: priority_queue &other)`: 把 `other` 合并到 `*this` 并把 `other` 清空。

使用的 tag 决定了每个操作的时间复杂度:

	push	pop	modify	erase	Join
`pairing_heap_tag`	$O(1)$	最坏 $\Theta(n)$ 均摊 $\Theta(\log(n))$	最坏 $\Theta(n)$ 均摊 $\Theta(\log(n))$	最坏 $\Theta(n)$ 均摊 $\Theta(\log(n))$	$O(1)$
`binary_heap_tag`	最坏 $\Theta(n)$ 均摊 $\Theta(\log(n))$	最坏 $\Theta(n)$ 均摊 $\Theta(\log(n))$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
`binomial_heap_tag`	最坏 $\Theta(\log(n))$ 均摊 $O(1)$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$
`rc_binomial_heap_tag`	$O(1)$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$
`thin_heap_tag`	$O(1)$	最坏 $\Theta(n)$ 均摊 $\Theta(\log(n))$	最坏 $\Theta(\log(n))$ 均摊 $O(1)$	最坏 $\Theta(n)$ 0 均摊 $\Theta(\log(n))$	$\Theta(n)$

```cpp
#include <algorithm>
#include <cstdio>
#include <ext/pb_ds/priority_queue.hpp>
#include <iostream>
using namespace __gnu_pbds;
// 由于面向 OIer, 本文以常用堆 : pairing_heap_tag 作为范例
// 为了更好的阅读体验，定义宏如下 :
#define pair_heap __gnu_pbds ::priority_queue<int>
pair_heap q1;  // 大根堆, 配对堆
pair_heap q2;
pair_heap ::point_iterator id;  // 一个迭代器

int main() {
  id = q1.push(1);
  // 堆中元素 :  [1];
  for (int i = 2; i <= 5; i++) q1.push(i);
  // 堆中元素 :  [1, 2, 3, 4, 5];
  std ::cout << q1.top() << std ::endl;
  // 输出结果 : 5;
  q1.pop();
  // 堆中元素 : [1, 2, 3, 4];
  id = q1.push(10);
  // 堆中元素 : [1, 2, 3, 4, 10];
  q1.modify(id, 1);
  // 堆中元素 :  [1, 1, 2, 3, 4];
  std ::cout << q1.top() << std ::endl;
  // 输出结果 : 4;
  q1.pop();
  // 堆中元素 : [1, 1, 2, 3];
  id = q1.push(7);
  // 堆中元素 : [1, 1, 2, 3, 7];
  q1.erase(id);
  // 堆中元素 : [1, 1, 2, 3];
  q2.push(1), q2.push(3), q2.push(5);
  // q1 中元素 : [1, 1, 2, 3], q2 中元素 : [1, 3, 5];
  q2.join(q1);
  // q1 中无元素，q2 中元素 : [1, 1, 1, 2, 3, 3, 5];
}
```

__gnu_pbds 迭代器的失效保证（invalidation_guarantee)

在上述示例以及一些实践中（如使用本章的 pb-ds 堆来编写单源最短路等算法），常常需要保存并使用堆的迭代器（如 `__gnu_pbds::priority_queue<int>::point_iterator` 等)。

可是例如对于 `__gnu_pbds::priority_queue` 中不同的 Tag 参数，其底层实现并不相同，迭代器的失效条件也不一样，根据 __gnu_pbds 库的设计，以下三种由上至下派生的情况:

1. 基本失效保证（basic_invalidation_guarantee）：即不修改容器时，点类型迭代器（point_iterator）、指针和引用（key/value）**保持**有效。
2. 点失效保证（point_invalidation_guarantee）：即 **修改**容器后，点类型迭代器（point_iterator）、指针和引用（key/value）只要对应在容器中没被删除 **保持**有效。
3. 范围失效保证（range_invalidation_guarantee）：即 **修改**容器后，除（2）的特性以外，任何范围类型的迭代器（包括 `begin()` 和 `end()` 的返回值）是正确的，具有范围失效保证的 Tag 有 rb_tree_tag 和适用于 `__gnu_pbds::tree` 的 splay_tree_tag (),以及适用于 `__gnu_pbds::trie` 的 pat_trie_tag。

从运行下述代码中看出，除了 `binary_heap_tag` 为 `basic_invalidation_guarantee` 在修改后迭代器会失效，其余的均为 `point_invalidation_guarantee` 可以实现修改后点类型迭代器 (point_iterator) 不失效的需求。

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

template <typename T>
void print_invalidation_guarantee() {
  typedef typename __gnu_pbds::container_traits<T>::invalidation_guarantee gute;
  cout << abi::__cxa_demangle(typeid(gute).name(), 0, 0, 0) << endl;
}

int main() {
  typedef
      typename __gnu_pbds::priority_queue<int, greater<int>, pairing_heap_tag>
          pairing;
  typedef
      typename __gnu_pbds::priority_queue<int, greater<int>, binary_heap_tag>
          binary;
  typedef
      typename __gnu_pbds::priority_queue<int, greater<int>, binomial_heap_tag>
          binomial;
  typedef typename __gnu_pbds::priority_queue<int, greater<int>,
                                              rc_binomial_heap_tag>
      rc_binomial;
  typedef typename __gnu_pbds::priority_queue<int, greater<int>, thin_heap_tag>
      thin;
  print_invalidation_guarantee<pairing>();
  print_invalidation_guarantee<binary>();
  print_invalidation_guarantee<binomial>();
  print_invalidation_guarantee<rc_binomial>();
  print_invalidation_guarantee<thin>();
  return 0;
}
```

(5). 可持久化数组/可持久化平衡树/块状链表 rope

要求引入头文件

```cpp
#include <ext/rope>
using namespace __gnu_cxx;
```

使用方法

```cpp
// 定义:
rope<int> rp;
```

成员函数

- `push_back(x)`: 在末尾插入 x
- `insert(pos, x)`: 在 pos 处插入 x
- `erase(pos, x)`: 在 pos 处删除 x 个元素

- `length()`: 返回数组长度
- `size()`: 返回数组长度 (同上)
- `replace(pos, x)`: 将 pos 处元素替换为 x
- `substr(pos, x, s)`: 从 pos 处开始提取 x 个元素
- `copy(pos, x, s)`: 从 pos 处开始复制 x 个元素到 s 中
- `at(x)`: 访问第 x 个元素，同 `rp[x]`

`rope` 内部是块状链表实现的，黑科技是支持 $O(1)$ 复制，而且不会空间爆炸 (`rope` 是平衡树，拷贝时只拷贝根节点就行)。因此可以用来做可持久化数组。

拷贝历史版本的方式:

```
rope<int> *his[100000];
his[i] = new rope<int> (*his[i - 1]);
```