map based on Size Balanced Tree

- 选题和背景
- 使用文档
- 结果展示
- 工作记录
- 缺点和不足

选题和背景

C++ 中，STL 中的 `std::set,std::map` 是非常常见的容器类型，它们提供了映射和集合的实现。我们熟知，这两种容器都并非线性存储数据，而是通过一棵二叉搜索树来存储数据，从而满足数据读写的速度要求。

为了保证数据读写的速度，两者均通过红黑树这个数据结构来存储数据，动态调整二叉搜索树的层高，从而达到 $O(\log n)$ 一次基础操作的时间复杂度。

但是传统的这两种容器具有一定的劣势：两者的迭代器均不支持随机访问，也就是说无法快速找到容器中的第 k 个内容、无法快速求出容器中两个位置的距离。这在逻辑上是可以理解的：集合和映射实质上是无序的，没有必要支持这两个操作。但是在实际使用中，很多时候会把这两个容器当作有序列表使用，对这两个功能的支持就有了意义。

因此，我们小组尝试利用另外的平衡树来维护集合中的数据，从而实现一个支持 $O(\log n)$ 随机访问的映射容器。

二叉平衡树选择

几十年中已经有了很多自平衡的二叉搜索树数据结构。我们希望找到一种结构：

- 支持节点的第 k 后继查询。
- 不需要在操作时产生太多额外空间。
- 性能较稳定，即树高较稳定。
- 即使效率不如红黑树，效率也不能过低。

最终我们选定了 Size-Balanced Tree ，这是一个基于子树大小来维持树高平衡的数据结构。由于在支持节点的 k 后继查询时，节点子树中节点个数，即 Size 是不能不维护的值，因此正好与 Size-Balanced Tree 的需要维护的内容重合。

Size-Balanced Tree 是中国人陈启峰在 2007 年提出的一种基于计算子树大小自平衡的数据结构，与 Weighted-Balanced Tree 比较类似，但是两者维护平衡的方式有一定出入。

Size-Balanced Tree 维护形态的方式也是旋转，因此在维护平衡时不会有过大的空间开销。且在陈的论文中证明了该树的高度 $O(\log n)$ 和各基本操作复杂度。

随机访问复杂度

假设在平衡树的每个节点维护了子树的大小，那么对 `map` 进行随机访问时，只需要在每个节点向左儿子或右儿子移动，同时维护当前仍需要向后多少即可。因此对 `map` 位置进行随机访问时需要的复杂度是 $O(h)$ 的，其中 $h = O(\log n)$ 为树高。

同理，在每个节点维护子树大小后，我们可以 $O(\log n)$ 得到一个位置的排名：即它向上移动并把经过的所有左子树大小求和即可。

这样可以在 $O(\log n)$ 的时间支持有序数列的随机访问、求距离等操作。

使用文档

基础使用方法和 `std::map` 一致，给出一些常用方法：

- `begin()`：返回指向 map 中第一个元素的迭代器。如果 map 为空，则返回尾后迭代器。
- `end()`：返回指向 map 中最后一个元素之后的迭代器。如果 map 为空，同样返回尾后迭代器。
- `find(const key_type& key)`：返回指向第一个关键字为 key 的元素的迭代器，如果 map 中不包含关键字为 key 的元素，则返回尾后迭代器。
- `insert(const value_type& val)`：将 val 插入到 map 中，并返回指向新元素的迭代器。
- `erase(iterator position)`：删除迭代器 position 指向的元素，并返回指向下一个元素的迭代器。
- `erase(const key_type& k)`：删除 key 值为 k 的元素，并返回指向下一个元素的迭代器。
- `clear()`：删除 map 中的所有元素。
- `size()`：返回 map 中元素的个数。
- `empty()`：如果 map 为空则返回 true，否则返回 false。
- `operator[]()`：提供对 map 中元素的访问，如果 map 中不存在相应关键字的元素，则会自动插入一个具有该关键字和一个默认值的元素。
- `count(const key_type& key)`：返回 map 中关键字为 key 的元素的个数，因为 map 中每个关键字都只能对应一个元素，因此返回值只能是 0 或 1。
- `lower_bound(const key_type& key)`：返回指向第一个关键字不小于 key 的元素的迭代器。如果不存在这样的元素，则返回尾后迭代器。
- `upper_bound(const key_type& key)`：返回指向第一个关键字大于 key 的元素的迭代器。如果不存在这样的元素，则返回尾后迭代器。
- `std::pair<iterator, bool> insert(value_type&& x)` 完美转发键值对，返回 `pair`
- `iterator insert(iterator pos, const value_type& x)` 在迭代器位置插入键值对
- `size_type erase(const key_type& k)` 寻找键为k的位置删除
- `iterator erase(iterator first, iterator last)` 范围删除

一些与 `std::map` 不同的点：

- 无法支持相同元素的插入，通过 `insert` 强行插入相同 Key 值的元素时行为未定义。

  逻辑上的 map 是不允许插入相同元素的，库文件中的红黑树为了支持 `multimap` 对相关部分做了明确处理，因此 map 实际上可以插入相同的元素。
- 没有一些 `emplace` 的支持。

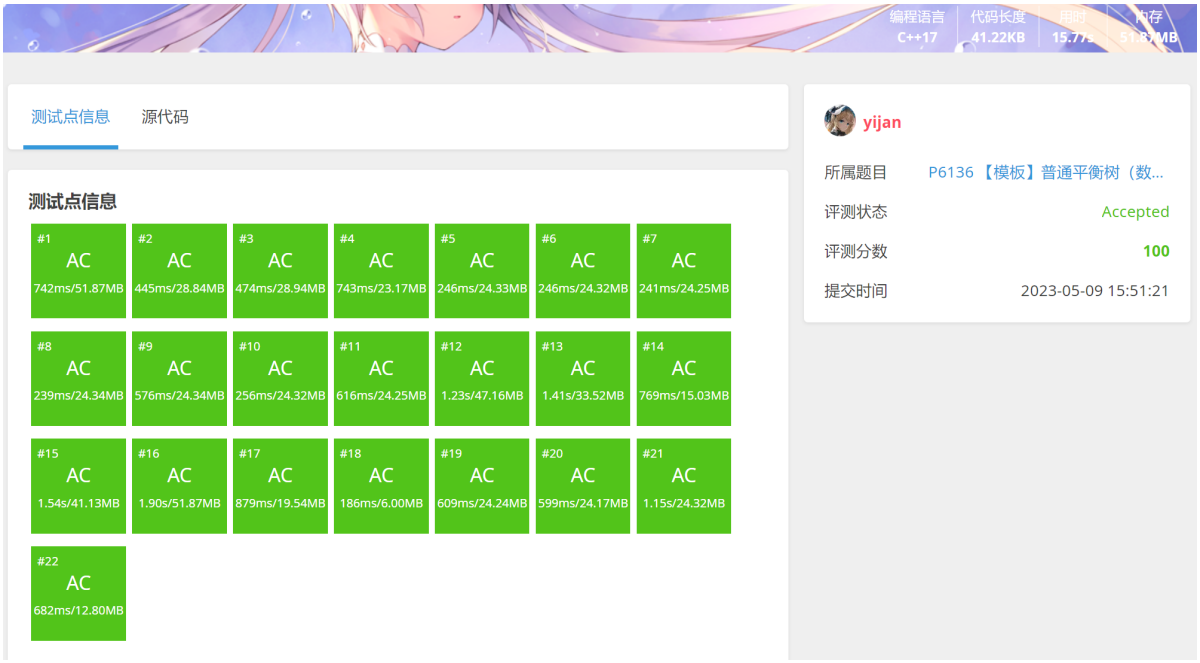
该 map 在迭代器上较 `std::map` 的迭代器实现了一些新功能：

- `iterator::operator +=( size_t __k )` 允许对一个迭代器向后移动 `__k` 位。在移动合法时等价于做 `__k` 次 `++` 操作
- `iterator::operator -= ( size_t __k )` 允许对一个迭代器向前移动 `__k` 位。在移动合法时等价于做 `__k` 次 `--` 操作。
- `iterator::operator + ( size_t __k )` 允许对一个迭代器访问其后 `__k` 个迭代器的位置。
- `iterator::operator - ( size_t __k )` 允许对一个迭代器访问其前 `__k` 个迭代器的位置。

- `size_t operator - ( const iterator& __x )` 允许对两个迭代器做差，得到两个迭代器的距离。
- `bool operator < ( const iterator& __k )` 比较两个迭代器在容器中的相对位置。

结果展示

我小组实现的平衡树可以通过 P6136 普通平衡树（加强版），且实际运行效果不差：



我们通过 Google Test 构造了一些测试数据与 `std::map` 对照来验证各功能的正确性：

```
[       OK ] size_balanced_tree.basic_info (3 ms)
[ RUN      ] size_balanced_tree.iterator
[       OK ] size_balanced_tree.iterator (0 ms)
[ RUN      ] size_balanced_tree.functions
[       OK ] size_balanced_tree.functions (3 ms)
[ RUN      ] size_balanced_tree.insert_small_int_1
[       OK ] size_balanced_tree.insert_small_int_1 (0 ms)
[ RUN      ] size_balanced_tree.copy_construct
[       OK ] size_balanced_tree.copy_construct (0 ms)
[ RUN      ] size_balanced_tree.copy_construct_map
[       OK ] size_balanced_tree.copy_construct_map (0 ms)
[ RUN      ] size_balanced_tree.basic_info
[       OK ] size_balanced_tree.basic_info (3 ms)
[ RUN      ] size_balanced_tree.iterator
[       OK ] size_balanced_tree.iterator (0 ms)
```

经过实际测试，该代码在时间消耗上大约是 `std::map` 的 $1.2 \sim 2$ 倍之间。

工作记录

基础

- 实现过程的代码规范问题参考 `stl_tree.h` 中红黑树的实现和 `CS106L` 中 `HashMap` 的实现。
- 目标为实现支持随机访问迭代器的平衡树类，在一般搜索二叉树的基础上额外维护每个子树大小，子树大小用 `unsigned int` 即 `size_t` 维护。
- `_Sb_tree_increment` 等函数用于访问节点的逻辑上的下一个节点（或上一个节点），不会产生任何异常，即通过 `noexcept` 标注。

- `_Sb_tree_increment` 或 `_Sb_tree_decrement` 可以给定参数 `__k` 表示访问向前或者向后的第 `__k` 个节点。
- 在定义平衡树时会给定从 `value_type` 中取出 `key` 的方法，这个方法的定义就是 `_KeyOfValue` 。
- 定义了一个 `_Node_allocator` 的派生类 `_Sb_tree_impl` 并定义 `_M_impl` 。其中包含继承而来的内存分配方式、整个树的比较方式、头以及树的大小。
- 在 `SB tree` 中只需要通过根节点的 `size` 就可以得到整个树的大小，没必要单独记录变量。
- 对 `_Sb_tree_impl` 给出一般、复制和移动构造函数。

节点构成与方法

- 利用 `_Sb_tree_node_base` 作为平衡树节点的基类，节点的左右儿子、父亲指针与子树大小均通过此类维护，平衡树节点在继承基类基础上维护节点对应值。
- 继承类拥有构造函数，从 `typename... _Args` 进行构造，具体构造方法为先调用 base 的构造函数，再通过 `_Args` 来构造继承类维护的 value 。通过 `std::forward` 进行转发。这个方法利用了 C++11 的特性，自由性高很多，可以给出 value 的任意一个构造函数的参数来对 value 进行构造。
- 对于头节点，其 `left,right` 表示最左和最右的节点，`parent` 表示整个树的根节点。头节点和根节点**互为** `parent` 关系，而头节点是 `_M_end()` ，根节点是 `_M_begin()` 。

  在外部实现 `increment,decrement,get_pos` 等操作时，很多时候需要判断一个点是否为根以及当前指针是否指向尾后指针，这个时候头节点（尾后指针位置）和根节点互为父亲关系就非常有效。

  为了方便进行判断，把头节点的大小设为根节点大小加一。
- 当整个树为空时，header 的 parent 为空，左右指针都指向本身。
- 对于找根、最左、最优、 `begin(),end()` 等操作，都同时需要有一般版本和 `const` 版本。
- 从节点中得到儿子、父亲、值等操作都是 `_rb_tree` 类的 `static` 函数。

迭代器

- 在 `iterator` 中 `->` 符号该如何被重载？事实上若 `->` 返回一个指针 `T*` ，那么 `p -> m` 可以被解释为 `(p.operator -> ()) -> m` 。
- 重载 `++,--` 时， `T operator ++()` 为 `++ x` 的重载， `T operator ++( int )` 为 `x++` 的重载，参数会被忽略。前者返回引用（左值），后者返回自加前的值（右值）。
- 我们需要定义一般 `iterator` 和 `const` 的 `iterator` ，分别实现功能。同时，对两者之间的相等和不等需要另外重载。
- 按道理来讲， `const iterator` 应当是不能转化为 `iterator` 的，但是实际上 `Rb_tree` 实现了这样一个方法。可能在后面会有使用。其中需要 `const_cast` 来丢掉 `const` 属性，这多数时候是非常危险的。

内存分配与释放

- `allocator` 是分配内存的方式类，默认为 `std::allocator` 。可以通过 `allocator<T>` 来得到一个对 `T` 类型分配内存的类。

  在实现中，用户可以对 `_Sb_tree` 指定自己的分配内存的方法，因此是一个模板参数。

  对 `_Val` 和 `node<_Val>` 的方法是一致的，因此需要 `_Alloc::template rebind<_Rb_tree_node<_Val>>::other` 来定义对 `node` 的分配内存方式。

*这里加 `template` 关键字是显示说明后面的 `dependent name` 是一个类模板。

*在模板中使用 `typename` 关键字修饰可以告知编译器后面的东西是一个嵌套类型而非一个变量。参考

- `get_allocator` 中直接使用了 `_M_get_Node_allocator` 并进行转换，这两者看起来并没有继承关系，且两者分配内存的类型是不同的，只通过 `rebind` 得到的另一个函数。
- 定义出创建和删除节点的方法，前者通过 parameter pack 来调用构造函数，后者先通过 Allocator 调用析构函数，再 deallocate 空间。前者会捕捉进行构造时的错误，如果发现错误则直接丢掉空间并抛出异常。
- 利用类 `_Alloc_node` 来分配节点（调用 `_M_create_node` ）。

杂项与疑问

- `_S_value` 仅用于比较值，因此 `_S_value` 的返回值总是个 `const_reference` 。
- 为了进行插入操作，需要先得到插入位置（一个 `pair` ，父亲和父亲对应指针）

  同时，由于在很多时候 `map` 中会先寻找一个位置，未找到再进行插入，需要实现 `_M_insert_hint_unqiue` 表示找到了需要插入的值的上一个或下一个位置时基于这个位置进行插入。这样可以减少再一次向下递归的时间。
- 目前不明白 `Is_pod_comparator` 在这里有什么用。（？）
- 没有实现 `range erase` ，只实现单次插入和单个迭代器删除。
- `lower_bound` 的标准写法：

  当当前节点大于等于需要寻找的 key 值时往左儿子走，并将另一个指针指向走之前的点。否则直接往右走。

  此时最终指针指向的值就是需要找到的值。
- 在 `tree.cpp` 中实现的本地函数需要加上 `static` 表示函数是 `local` 使用的。
- `delete` 实现中首先是正常平衡树删除，然后从最下的影响节点往上进行平衡和 `size` 更新。
- 通过 `size` 的维护可以做到 $O(\log n)$ 时间进行随机访问。对于一个位置，如果想要找到 `p + k` 只需要先不断向上，每次给 k 减少右儿子子树大小，直到右子树大小小于这个值，然后向右子树走，再进行查找即可。
- 通过 `size` 的维护可以 $O(\log n)$ 时间进行排名查询。以此可以做到 $O(\log n)$ 时间的 `distance` 实现。

缺点和不足

相对于红黑树实现的 `std::map` ，该方法虽然能够支持更快的随机访问，但是也带来一些损失：

- 时间消耗相对更大。

  在实际测试中，多数操作会比 `std::map` 中的红黑树效率更低。
- 空间开销更大。

  红黑树中每个节点只需要存储一个 `enum` 作为颜色，只有两种取值，以及左右儿子和父亲指针。该方法实现平衡树时需要每个点的子树大小，这会导致更大的空间开销。
- 在实现过程中，对库函数的某些处理仍然存在疑问

  没能完全明白库函数的某些处理手段和技巧的优势，只能体会到某些实现的精妙。
- 测试强度不足。

由于没有时间去构造更强、更全面、更广泛的数据来对 `map` 进行测试，可能仍然存在一定的 Bug 未被发现。