

数据结构课程项目实验报告

杜雨轩 19307130196

0.目录

- 整体介绍
- 结构说明
- 具体功能及时间复杂度分析
- 优势与劣势分析
- 测试
- 心得体会
- 参考资料

一.整体介绍

- 本数据结构以红黑树为基础，以较好的平均性能实现了课程项目的基本操作
- 本数据结构很重视接口的规范，在设计时相对严格地遵守了STL的设计规范

二.结构说明

以下为各类、结构的数据段内容

- 节点 Node
 - 父母节点的指针
 - 左右子女节点的指针
 - 值
 - 以该点为根的子树大小
 - 节点颜色
- 内存池 MemoryPool
 - 节点数组
 - 指向节点数组中第一个未使用节点的指针
 - 资源回收栈
 - 资源回收栈的栈顶指针
- 迭代器 MyIterator
 - 指向节点的指针
- “线性表” LinearTable
 - 一个节点 (header, 起到控制器的作用)
 - 容器当前元素总数

各类、结构中有一些typedef，详见源码，此处不多作介绍了。

类的成员函数会在稍后作详细介绍。

三.具体功能及时间复杂度分析

构造函数，析构函数

函数定义	函数说明	时间复杂度 (平均)	空间复杂度 (指额外空间)	备注
LinearTable()	默认构造函数	O(1)	O(1)	
~LinearTable()	析构函数	O(N): 遍历	O(log N): 递归的栈空间	空间可以改进至O(1)
LinearTable(size_type n, value_type value = 0)	构造函数, 插入多个相等值	O(nlogn)	O(1)	朴素做法
LinearTable(const_pointer first, const_pointer last)	构造函数, 沿指针插入	O(nlogn)	O(1)	朴素做法, 只支持指针
LinearTable(const_iterator first, const_iterator last)	构造函数, 沿常量迭代器插入	O(nlogn)	O(1)	朴素做法, 只支持本容器
LinearTable(iterator first, iterator last)	构造函数, 沿迭代器插入	O(nlogn)	O(1)	朴素做法, 只支持本容器
LinearTable(const LinearTable& other)	复制构造函数	O(n)	O(1)	deep-copy
LinearTable& operator=(const LinearTable& other)	等号重载	O(n)	O(1)	deep-copy
LinearTable(LinearTable&& other)	移动构造函数	O(1)	O(1)	shallow-copy 配合 std::move

基础函数(此段n代表容器中的元素个数，m代表迭代器之间的差值)

函数定义	函数说明	时间复杂度(平均)	空间复杂度(指额外空间)	备注
iterator begin()	获取首迭代器	$O(1)$	$O(1)$	
const_iterator begin()	获取常值首迭代器	$O(1)$	$O(1)$	
iterator end()	获取尾迭代器	$O(1)$	$O(1)$	
const_iterator end()	获取常值尾迭代器	$O(1)$	$O(1)$	
reference front()	获取首个元素的引用	$O(1)$	$O(1)$	
const_reference front()	获取首个元素的常值引用	$O(1)$	$O(1)$	
reference back()	获取末尾元素的引用	$O(1)$	$O(1)$	
const_reference back()	获取末尾元素的常值引用	$O(1)$	$O(1)$	
size_type size()	获取容器当前节点个数	$O(1)$	$O(1)$	
size_type max_size()	获取容器最大节点数	$O(1)$	$O(1)$	
bool empty()	判断容器是否为空	$O(1)$	$O(1)$	
void swap(LinearTable& other)	交换两个容器	$O(1)$	$O(1)$	
void clear()	清空容器	$O(1)$	$O(1)$	
reference operator[](size_type pos)	获取某位置引用	$O(\log n)$	$O(1)$	
const_reference operator[](size_type pos)	获取某位置常值引用	$O(\log n)$	$O(1)$	
iterator PubkthIt(size_type k)	获取某位置的迭代器	$O(x \log n)$	$O(1)$	自定义工具函数
void push_back(value_type value)	尾插入	$O(\log n)$	$O(1)$	
void push_back(size_type x, value_type value)	尾插入x次	$O(\log n)$	$O(1)$	
void push_back(const_pointer first, const_pointer last)	尾插入，以指针	$O(m \log (n+m))$	$O(1)$	
void push_back(const_iterator first, const_iterator last)	尾插入，以常值迭代器	$O(m \log (n+m))$	$O(1)$	仅限本容器迭代器

函数定义	函数说明	时间复杂度(平均)	空间复杂度(指额外空间)	备注
void push_back(iterator first, iterator last)	尾插入，以迭代器	$O(m \log(n+m))$	$O(1)$	仅限本容器迭代器
void pop_back()	尾删除	$O(\log n)$	$O(1)$	
void pop_back(size_type x)	尾删除x次	$O(x \log n)$	$O(1)$	
void push_front(value_type value)	首插入	$O(\log n)$	$O(1)$	
void push_front(size_type x, value_type value)	首插入x次	$O(x \log n)$	$O(1)$	
void push_front(const_pointer first, const_pointer last)	首插入，以指针	$O(m \log(n+m))$	$O(1)$	
void push_front(const_iterator first, const_iterator last)	首插入，以常值迭代器	$O(m \log(n+m))$	$O(1)$	仅限本容器迭代器
void push_front(iterator first, iterator last)	首插入，以迭代器	$O(m \log(n+m))$	$O(1)$	仅限本容器迭代器
void pop_front()	首删除	$O(\log n)$	$O(1)$	
void pop_front(size_type x)	首删除x次	$O(x \log n)$	$O(1)$	

插入删除(此段n代表容器中的元素个数，m代表迭代器之间的差值)

函数定义	函数说明	时间复杂度 (平均)	空间复杂度(指 额外空间)	备注
iterator insert(iterator position, const value_type& value)	在给定迭代器前插入元素，返回插入元素的迭代器	$O(\log n)$	$O(1)$	
iterator insert(iterator position, size_type x, const value_type& value)	在给定迭代器前插入x个相同元素，返回插入首个元素的迭代器	$O(x \log n)$	$O(1)$	朴素做法
iterator insert(iterator position, const_pointer first, const_pointer last)	在给定迭代器前插入系列元素，返回插入首个元素的迭代器，以指针	$O(x \log n)$	$O(1)$	朴素做法
iterator insert(iterator position, const_iterator first, const_iterator last)	在给定迭代器前插入系列元素，返回插入元素的迭代器，以常值迭代器	$O(x \log n)$	$O(1)$	朴素做法
iterator insert(iterator position, iterator first, iterator last)	在给定迭代器前插入系列元素，返回插入元素的迭代器，以迭代器	$O(x \log n)$	$O(1)$	朴素做法
iterator erase(iterator position)	擦除给定迭代器所指元素，返回下一个迭代器	$O(\log n)$	$O(1)$	
iterator erase(iterator first, iterator last)	擦除给定迭代器区间的所有元素	$O(x \log n)$	$O(1)$	朴素做法

迭代器操作(此段n代表容器中的元素个数，m代表迭代器之间的差值)

函数定义	函数说明	时间复杂度(平均)	空间复杂度(指额外空间)	备注
reference operator*()	取值的引用	O(1)	O(1)	
pointer operator->()	取指针	O(1)	O(1)	
self& operator++()	自增后返回	O(1)	O(1)	
self operator++(int)	自增, 返回原迭代器	O(1)	O(1)	
self& operator--()	自减后返回	O(1)	O(1)	
self operator--(int)	自减, 返回原迭代器	O(1)	O(1)	
self& operator+=(int x)	返回后x个迭代器	O(log n)	O(log n)	x可正可负
self& operator-=(int x)	返回前x个迭代器	O(log n)	O(log n)	x可正可负
reference operator[] (const unsigned long long x)	返回后x个迭代器对应的值	O(log n)	O(log n)	x可正可负 (符合stl标准)
bool operator==(const MyIterator& other)	判断迭代器是否相等	O(1)	O(1)	
bool operator!=(const MyIterator& other)	判断迭代器是否不相等	O(1)	O(1)	
bool operator<(const MyIterator& other)	判断迭代器是否在给定迭代器前	O(log n)	O(log n)	
bool operator>(const MyIterator& other)	判断迭代器是否在给定迭代器后	O(log n)	O(log n)	
bool operator<=(const MyIterator& other)	判断迭代器是否在给定迭代器前或相等	O(log n)	O(log n)	
bool operator>=(const MyIterator& other)	判断迭代器是否在给定迭代器后或相等	O(log n)	O(log n)	
MyIterator operator+(MyIterator it, int x)	计算it后第x个迭代器	O(log n)	O(log n)	x可正可负
MyIterator operator+(int x, MyIterator it)	计算it后第x个迭代器	O(log n)	O(log n)	x可正可负
MyIterator operator-(MyIterator it, int x)	计算it前第x个迭代器	O(log n)	O(log n)	x可正可负

函数定义	函数说明	时间复杂度(平均)	空间复杂度(指额外空间)	备注
int operator-(MyIterator it1, MyIterator it2)	计算it1在it2前几位	$O(\log n)$	$O(\log n)$	结果可正可负

区间最值操作(此段n代表容器中的元素个数，m代表迭代器之间的差值)

函数定义	函数说明	时间复杂度 (平均)	空间复杂度 (指额外空间)	备注
value_type max(iterator first, iterator last)	求最大区间值	O(m)	O(1)	遍历
value_type max(const_iterator first, const_iterator last)	求最大区间值	O(m)	O(1)	遍历
value_type min(iterator first, iterator last)	求最小区间值	O(m)	O(1)	遍历
value_type min(const_iterator first, const_iterator last)	求最小区间值	O(m)	O(1)	遍历
value_type min(iterator first, iterator last, size_type k)	求区间的k小	O(m)	O(m)	基于 std::nth_element
value_type max(iterator first, iterator last, size_type k)	求区间的k大	O(m)	O(m)	基于 std::nth_element
value_type min(const_iterator first, const_iterator last, size_type k)	求区间的k小	O(m)	O(m)	基于 std::nth_element
value_type max(const_iterator first, const_iterator last, size_type k)	求区间的k大	O(m)	O(m)	基于 std::nth_element
std::vector<value_type> min(iterator first, iterator last, size_type lbound, size_type rbound)	求区间的 [lbound,rbound) 小	O(mlogm)	O(m)	基于std::sort
std::vector<value_type> max(iterator first, iterator last, size_type lbound, size_type rbound)	求区间的 [lbound,rbound) 大	O(mlogm)	O(m)	基于std::sort
std::vector<value_type> min(const_iterator first, const_iterator last, size_type lbound, size_type rbound)	求区间的 [lbound,rbound) 小	O(mlogm)	O(m)	基于std::sort

函数定义	函数说明	时间复杂度 (平均)	空间复杂度 (指额外空间)	备注
<code>std::vector<value_type> max(const_iterator first, const_iterator last,size_type lbound,size_type rbound)</code>	求区间的 [lbound,rbound) 大	$O(m\log m)$	$O(m)$	基于std::sort

有序数组操作(此段n代表容器中的元素个数，m代表迭代器之间的差值)

函数定义	函数说明	时间复杂度 (平均)	空间复杂度(指额外空间)	备注
<code>void sort(iterator first, iterator last)</code>	区间升序排序	$O(m\log m)$	$O(m)$	基于std::sort
<code>iterator sorted_insert(const value_type& value)</code>	向升序容器插入元素	$O(\log n)$	$O(1)$	
<code>void sorted_merge(const LinearTable& other)</code>	合并2个升序容器	$O(n\log n)$	$O(1)$	类似归并

四.优势与劣势分析

优势

相较于std::deque在中间插入删除 $O(n)$ 的低性能，std::list在随机访问 $O(n)$ 的低性能，我基于红黑树实现的LinearTable均可在 $O(\log n)$ 的时间内完成任务。更准确的说，除了最值、清空、批量插删、深拷贝、有序容器这几类操作，均能做到 $O(\log n)$ 或 $O(1)$ 的复杂度，这种复杂度反映了这个容器的全能性，综合表现性能好。尤其在同时要求中间插删与随机访问的领域有着良好的性能。

红黑树在诸多平衡树中，常数相对较小，因此在同类型以平衡树为原型的数据结构实现中表现较好。

数据结构设计时参考了sgi-stl/stl_tree.h的设计，因此设计相对规范，支持随机访问迭代器的各种表达式，支持stl容器的各种构造形式(包括移动构造)，支持常量容器与常量迭代器。

设计了内存池，在保证容器可变长的情况下，避免多次调用new和delete，理论上会有更好的性能。

劣势

全能往往是以失去特长为代价的，在单项功能的实现中，这个容器往往不能做到最佳，头尾插入删除、随机访问不如std::deque，中间插入删除不如std::list这是明确的，所以在某些测试中表现略差。

红黑树结构比较复杂，细节极多，可能会有一些错误(目前遇到的均已解决)

五.测试

本容器通过oj上除"min_2"外所有测试*(截至2021/1/1), 以及在洛谷上其他同学的测试数据中也全数通过。并且, 在Base_Operation_Plus这一测试中与其他树形结构实现具有相对明显的优势, 在min上也有速度优势(虽然我认为是很多同学不知道nth_element的O(n)实现的原因)。在construct_test上体现了容器规范与标准。

*alloc_test_2我的实现方式有些许问题, oj上可以通过, 但在本地gcc8.1.0及洛谷上编译错误。

我参与了oj上hack数据的创作, 有测试构造函数与内存分配器的2个数据。

我还在洛谷上提交了一个有难度梯度的中间插入与删除测试(另外还需要支持迭代器的随机访问it+n), 随机生成数据, 与std::deque对拍结果完全一致且效率更高。

我还为各种操作写了一个简短的正确必要性测试。

以上测试我会整理到附件文件夹中。

OJ测试情况如下

2021-1-3 10:16:29	87f0d651c55a	Accepted	Constructor_1	1ms	3MB	C++	19307130196
2021-1-3 10:16:16	879e3426f143	Accepted	Base_Operation_0_5	51ms	4MB	C++	19307130196
2021-1-3 10:15:49	d0650d9ddc77	Accepted	Base_Operation_Plus	5323ms	8MB	C++	19307130196
2021-1-3 10:15:32	d939292a378d	Accepted	Base_Operation	1ms	3MB	C++	19307130196
2021-1-3 10:17:19	4bb47b58c74f	Accepted	alloc_test_1	5204ms	44MB	C++	19307130196
2021-1-3 10:17:04	3eea945ab4f3	Accepted	batch_push_pop_1	2172ms	284MB	C++	19307130196
When	ID	Status	Problem	Time	Memory	Language	Author
2021-1-3 10:24:08	34efe6bf7401	Accepted	min_3	351ms	4MB	C++	19307130196
2021-1-3 10:23:38	95d5af568636	Accepted	min_1	11156ms	6MB	C++	19307130196
2021-1-3 10:39:27	d04a91a0c242	Accepted	树形结构随机访问测试	10499ms	461MB	C++	19307130196

//树形结构随机访问原本数据量级1e7, 时限10000ms, 此处为请助教开大时限测的一个值。

2021-1-3 10:25:52	9c9167241f2e	Accepted	foreach_test	0ms	3MB	C++	19307130196
2020-12-30 11:10:18	cfbd2235a677	Accepted	alloc_test_2	2ms	3MB	C++	19307130196

//alloc_test_2 我用了泛型, 虽然oj能过, 但本地编译不过, 待解决。

**mazymaze** ✓
01-01 10:46:41

Accepted
100

U147036 数据结构PJ-中间插删

🕒 1.31s / 📦 15.36MB / 📄 33.29KB C++14 🗑️ 2

这是我自己做的中间插删数据, 截至2021/1/3/10:55最快(之一)。

六.心得体会

终于写到这里了,, , 允许我不那么严肃吧。选红黑树的原因? 一开始我希望写一个全能的容器, 所以就觉得用平衡树作为基础, 去实现这个功能复杂的容器吧。红黑树有比较快, 代码难写一点也不是不能接受。开头那么几周还是几天还在想怎么整索引的事情, 但后来清醒了, 明白了我们索引和现在有的那些借口比如push_front等, 本质是一样的, 都是对指针移动行为的一种控制。想清楚后完成初版, 却发现无法通过大样例, 想来想去找不出问题干脆重构。这次重构借鉴了stl_tree.h, 那里有比较好的红黑树实践。但是, 代码是不能照搬的。我在理解他的实现后, 写出了另外一棵树, 需要维护多一个size域, 以及实现随机访问迭代器的功能。还是出了几个bug。众所周知, 树形结构调试困难, 所以有的时候就陷入绝望或困惑, 盯着代码看了好久而无头绪。很庆幸最后还是修好了。但是树形结构可能是真的恶心。。。

还是有所收获的，我重新理了一遍红黑树的逻辑，(虽然很可能很久又用不上了)，也体验了一下stl的编写工作，确实相当地困难。。主要还是心态上的进步吧，更加不畏困难了？希望如此吧。

七.参考资料

- stl_tree.h, STL, Cygnus C++ 2.91 for windows
- 《STL源码剖析》by 侯捷
- 《浅谈红黑树》by bf , <https://www.luogu.com.cn/blog/bfqag/qian-tan-hong-hei-shu>