结合动态顺序统计树与deque的线性表实现

2020-Fall Term DataStucture PJ

摘要

本次 PJ 要求实现一个外部接口类似 deque 的数据结构，包括头尾插删，中间插删等。尽量做到更优的复杂度。前期准备过程中，我尝试了各种数据结构的可能性，包括纯粹的 deque 实现， 基于 key 值的平衡二叉搜索树的线性表套壳实现，最终或是因为复杂度原因，或是因为 key 值维护的困难性，都被我否定了。

在此之后我便在思考，对于线性结构，其拥有较好的头尾插入性能 $O(1)$，但其中间插入却是其劣势所在 $O(n)$，相比下来树形结构能够做到更优的中间插入性能，但相对之下却牺牲了首尾插入的性能 $O(lgn)$，我便试想能否将**线性结构和树形结构进行结合**，实现二者的平衡呢?

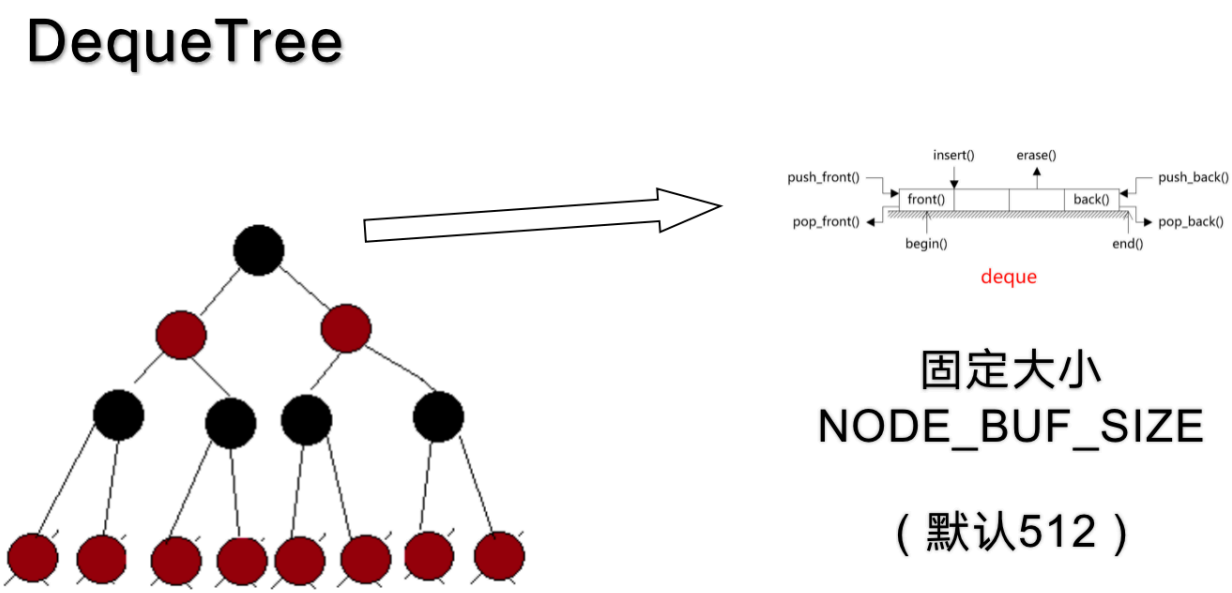
于是我通过基于**动态顺序统计树和线性deque相结合**的方式，实现线性表。实验表明，这种数据结构可以很好的在**头尾插入和中间插入之间寻求平衡**，做到较好的复杂度。我将其称为 DequeTree。它可以通过**动态调整** NODE_BUF_SIZE 来调整其性能表现和各类操作的负责度

(本文中将动态顺序统计树简称为红黑树)

关键词： 树形线性结合，头尾中间插入性能，DequeTree，动态调整

目录

简介



我的数据结构的基本模式如上图所示。其主体结构是一颗红黑树，准确的说是动态顺序统计树，其上每一个节点中，存储的并不是一个值，而是一个固定长度的 deque，这一固定长度我称之为 NODE_BUF_SIZE 默认我设置为 512。

我们知道限制线性结构中间插入删除的是连续紧密的内存储存，导致其必须将后面的元素全部移动后才能进行插入，而对于树形结构则可以直接通过树链找到插入位置直接插入消耗的只是在树上查找的时间 lgn (实际上在储存了 front_node 和 back_node 后在树上的插入复杂度是维护size的时间为 lgn 后文会进行详解)。而限制树形的头尾插入的同样是这个树链移动维护的时间，相比之下线性结构可以直接实现 $O(1)$的首尾插入。

对于我这样的树形和线性结合的数据结构，它的重大意义是通过增大每个节点的数据量，降低了树高，从而减少了树上查找的时间，同时因为内部是线性结构，在首尾插入时仍然是 $O(1)$ 的复杂度，在中间插入时由于其大小固定，不需要像普通线性表一样移动所有的元素，只要移动少部分即可且是动态的。所以依然可以保证较好的头尾和中间插入性能。

关于动态顺序统计树

我的数据结果是基于动态顺序统计树的基础上完成的，接下来我将会介绍我是怎么通过一系列操作使动态顺序统计树满足我们的要求。首先我们要实现一个线性表，首要需要解决的就是它的顺序访问问题，如果不能够进行按插入顺序进行访问，那么线性表的其他操作就无从谈起。

顺序访问实现的可能性

如果我们手上拿到一颗普通的红黑树，例如 stl 的 map ，为了解决这一问题我们很自然的会想到用插入的顺序作为一个 key 值来维护其顺序，访问时只要对 key 值和插入顺序下标进行一个映射操作就好，虽然这样的方式能够暂时解决我们的顺序访问问题，但其存在着巨大的bug。例如在中间插入时，由于 key 值是紧密排列的整型，我们在中间插入就必然要对后续的节点的key值进行统一的维护，这就又导致了其插入性能变成了 $O(n)$ 甚至更差。同时如果我们想方设法将 key 搞得再稀疏，或者定期进行重新映射。在大量中间插入的时候这种行为都是无法接受的，它将完全丢失红黑树中间插入的优势。

所以我们显然不能够通过储存 key 值来实现下标访问，那实际上要实现树上的随机访问，有一种数据结构称为动态顺序统计树，在《算法导论》的第十四章中有所提及，对于这种数据结构，其本身是一颗红黑树，不同的是其每个节点多储存一个 size 信息，即为其子树的大小加上它自身的大小。那么这样我们就可以通过 `cur_node->left->size + 1` 的形式访问到当前节点的中序遍历位置，通过这一方法我们可以实现按照树内部中序顺序的顺序访问，我们称节点中序的位置 `cur_node->left->size + 1` 为该节点的秩 `rank` 。

- **Rank 的提出**

具体做法就是从根节点出发，比较 `key` 和 `rank` 的大小，如果小于说明在左子树中，那么我们从树链走到左子节点再重复操作即可；如果大于说明在右子树中，这时我们在树链上走到右节点上，同时将 `key` 减去 `rank` ，再重复操作，这样下去我们最终可以找到 `rank == key` 的节点。

- **自定义插入位置**

但这样就要求我们保证树的中序遍历顺序就是插入的顺序，很幸运，我们知道红黑树对于平衡顺序的维护并不改变树的中序遍历的顺序且和节点的值本身没什么关系（二叉搜索树节点值的有序本质上是插入过程保证的），所以我们的红黑树未必一定是一颗二叉搜索树，只要按照对应的要求插入之后进行 `InsertFix` 或者 `DelFix` 即可，这为我们的实现提供了可能性。我们可以直接控制插入的位置，而不用像传统红黑树或传统动态顺序统计树一样按照二叉搜索树的值比较找到插入位置再进行插入。这样就形成了我们特定的数据结构模型

- **对于 DequeTree 顺序访问的修改**

  对于 DequeTree 由于每一节点不仅仅储存一个值，所以相对应的 size 也要进行修改，我在维护了普通动态顺序统计树的 size 同时也对每个节点维护一个真实的数据量 sz 它表示了该节点子树的数据量加上该节点本身的数据量。同时我们对 `key` 的查询也要进行修改 我定义了 `rank_low = cur_node->left->sz` 和 `rank_high = cur_node->left->sz + cur->data.size()` 来查询 key 具体的树链查找思路不变，最后只要有 `key > rank_low && key <= rank_high` 我们便知道要找寻的值落在了改节点中，之后再用 `cur->data[key - cur->left->sz - 1]` 将对应的值 $O(1)$ 取出即可。

Size 和 Sz 的修复

动态顺序统计树相对于普通红黑树的插入除了插入的红黑性质修复以外，同时要维护 size，这一过程就是从插入节点沿着树链往上走到根节点把走到节点的 size 都 +1 即可。而在左右旋的过程中由于树的结构发生了改变，size也要做出相应的改变，在左右旋完成后对节点进行 size 的重新统计

```
y->size = x->size;
x->size = x->left->size + x->right->size + 1;
```

相比之下对于 DequeTree 同时也要维护 sz 在维护 size 的过程中我们将 ±1 对应改为 ± cur_node->data.size() 即可

- 维护 Size 和 Sz 的函数

对于 Size 和 Sz 的维护我封装了四个非常类似的函数，分别是 `FixSizeInc`，`FixSizeDec`，`FixSzInc`，`FixSzDec`。

```
// size和sz的维护为了保证数据不溢出，所以增减各写了一个
    void FixSizeInc(RB_Node*, const size_type& n); // 将size++, sz += n, 剩下逻辑类似
    void FixSizeDec(RB_Node*, const size_type& n);
    void FixSzInc(RB_Node*, const size_type& n);
    void FixSzDec(RB_Node*, const size_type& n);
```

插入的优化

我们又注意到对于首尾插入明显是可以进行优化的，可以通过将 front_node 和 back_node 储存在表中，同时在操作过程中动态维护的方式，以在首尾插入的过程中 O(1) 的获得首尾节点，但该优化对于树高较高的红黑树可能有一定作用，但对于 DequeTree 因为本身树高较小，收效甚微，甚至在某些数据量范围下由于动态维护的特判，导致性能下降的可能。经测试在 Base_OP_Plus 1e7 的数据量下优化了 100ms 左右 。

复杂度分析

本节主要进行功能的复杂度分析

由于 DequeTree 通过 NODE_BUF_SIZE 很好的降低了树高，所以在复杂度方面，表现出了均衡的性能。

复杂度分析	Deque	RBTree	DequeTree
随机访问	$O(1)$	$O(lgn)$	$O(lg(\frac{n}{x}))$
首尾插删	$O(1)$	$O(lgn)$	$O(lg(\frac{n}{x}))$
中间插删	$O(n)$	$O(lgn)$	$O(lg(\frac{n}{x})+x)$
首尾批量删插	$O(m)$	$O(mlgn)$	$O(mlg(\frac{n}{x}))$
中间批量删插	$O(mn)$	$O(mlgn)$	$O(mlg(\frac{n}{x})+mx)\approx O(mlg(\frac{n}{x}))$

（上表中 x 代表 NODE_BUF_SIZE）

我们可以通过动态调整 NODE_BUF_SIZE 来对我们的数据结构进行动态调整

关于 NODE_BUF_SIZE

如果了解过 deque 底层我们都应该知道， deque 底层初始是由一个 大小为8的 map 空间指向8个 BUF_SIZE 为 512 的数组所拼接而成的结构，如果插入数据大于 8 * BUF_SIZE 那么我们就将其复制移动到一个更大的内存空间中去。所以理论上对于 deque 小于 512 的插入删除，其实跟数组没什么区别，对于 NODE_BUF_SIZE 的动态调整，我建议最高上限 不超过 2^{12} 即 8 * NODE_BUF_SIZE 的上限，不然会触发 deque 的内存扩展导致效率下降。（当然也可以在初始化 deque 的时候指定其 NODE_BUF_SIZE 的大小来进行动态调整）同时最低不低于 2^4 即 16 的单个deque大小，如果进一步变小 deque 展现的头尾插入优化便极其细微，同时因为结构的复杂会导致常数巨大。

- **DequeTree 与 RBTree 对比**

值得注意的是, DequeTree 在首尾插入和中间插入性能中做了平衡，DequeTree 的中间插入总是慢于红黑树的中间插入 $lgn < lg(\frac{n}{x})+x$, 即使把NODE_BUF_SIZE 调整为 1 但由于结构的复杂性和常数其中间插入仍慢于优化好的红黑树, 但其速度显然快于 Deque 的中间插入。同时红黑树由于节点众多会导致在不开内存池情况下频繁内存分配导致的效率下降，以及大数据量下极慢的析构过程。而 DequeTree 的析构则明显更快

- **对于中间插入 NODE_BUF_SIZE 的调整**

我们可以通过动态的调整 NODE_BUF_SIZE 来做到更优的适应使用场景和数据量大小，相关测试请见**性能测试**节

功能详解

下面我将对我实现的功能和代码进行详解。

TypeDeclare

在开始介绍之前，先介绍一下我定义的内部类型，分别定义了 value_type, key_type, node_type 等类型

```cpp
// Type Declare
typedef int value_type;
typedef long long key_type;
typedef std::deque<value_type> node_type;
typedef value_type* node_ptr;
typedef std::ptrdiff_t difference_type;
typedef std::size_t size_type;
```

构造函数

构造函数我实现了多种构造类型，主要种类如下 (细节太多了，详见代码)

```cpp
LinearTable();  // 默认构造
LinearTable(const LinearTable&); // 复制构造函数 深拷贝
LinearTable(LinearTable&&); // 移动构造函数
LinearTable(const int&, const int&); // 初始化 n 个 val 的构造函数
template <typename _InputIter>
LinearTable(_InputIter a, _InputIter b);   // 迭代器构造
~LinearTable();
```

内部维护 private 接口

我还定义了一系列用于内部维护的接口，这边仅进行展示，细节详见代码

```cpp
void RotateLeft(RB_Node*);
void RotateRight(RB_Node*);
RB_Node* lNode(RB_Node*);   // 树链最左
RB_Node* rNode(RB_Node*);   // 树链最右
RB_Node* Predecessor(RB_Node*);
RB_Node* Successor(RB_Node*);

void InsertFix(RB_Node*);
void DelFix(RB_Node*);

// 可考虑内存池优化
RB_Node* CreateNode(const value_type&);
RB_Node* CreateNode(const size_type&, const value_type&);

void Transplant(RB_Node*, RB_Node*);    // 把第二个参数的子树种到第一个参数上去
void PushBackNode(RB_Node*, const size_type&);
void PushFrontNode(RB_Node*, const size_type&);
void DelNode(RB_Node*);

RB_Node* FindNode(key_type);    // 根据 key 找到对应节点
RB_Node* backNode();
RB_Node* frontNode();

void DelTree(RB_Node*);
```

外部接口

接下来我将详细解释外部用户可调用的方法，主要内容如下

```
public:
    // 外部开放 api
    void push_back(const value_type&);
    void push_back(size_type, const value_type&);
    void push_front(const value_type&);
    void push_front(size_type, const value_type&);

    void pop_back();
    void pop_back(size_type);
    void pop_front();
    void pop_front(size_type);

    void clear();
    void swap(LinearTable&);

    size_type size();
    bool empty();

    value_type front();
    value_type back();


    value_type& operator[](key_type);
    const value_type& operator[](key_type) const;
    value_type& at(key_type);
    value_type modify(key_type, const value_type&);

    // ------- iterator extended -------- //

    iterator begin();
    const iterator begin() const;
    iterator end();
    const iterator end() const;

    iterator insert(iterator, const value_type&); // 在 iterator 前 插入一个 val
    iterator insert(iterator, size_type, const value_type&); // 在 iterator 前 插入 size 个 val

    iterator erase(iterator); // 删除迭代器位置的元素
```

- **随机访问**

随机访问的方法在前文中解释的已经比较清楚，就不再赘述，此处放一下代码，方便阅读

```
inline value_type& LinearTable::operator[](key_type key)
{
    key++; // 1 ~ n
    RB_Node* cur = root;
    size_type rank_low, rank_high;
    while(cur != nil)
    {
        rank_low = cur->left->sz;
        rank_high = cur->left->sz + cur->data.size();
        if(key > rank_low && key <= rank_high)  // 在中间说明就在该节点中
            return cur->data[key - cur->left->sz - 1];
        else if(key <= rank_low) // 如果小在左边
```

```
            cur = cur->left;
        else  // 大就往右走
        {
            cur = cur->right;
            key -= rank_high;
        }
    }
    throw std::domain_error("Segmentation Fault, Cannot find the key!\n");
}
```

- **首尾插删类**

以首尾插入为例，删除同理。

```
void push_back(const value_type&);
void push_back(size_type, const value_type&);
void push_front(const value_type&);
void push_front(size_type, const value_type&);
```

这四个方法，代表首尾单插和批量插入

```
void LinearTable::push_back(const value_type& val)
{
    if(empty())
    {
        root = CreateNode(val);
        root->color = BLACK;
        front_node = root;
        back_node = root;
    }
    else if(back_node->data.size() >= NODE_BUF_SIZE) // 如果超节点上限了就新创一个节点
    {
        RB_Node* newNode = CreateNode(val);
        PushBackNode(newNode, 1);
    }
    else // 如果没超上限直接后端插入
    {
        FixSzInc(back_node, 1);
        back_node->data.push_back(val);
    }
}
```

以单个 push_back 为例，分为表为空，尾部节点有空，尾部节点没空三种情况。然后对于尾部节点没空的情况，我们新插入一个节点，此处我为了方便，封装了一个 PushBackNode 方法，在其中插入了尾部节点，并且对 size 和 sz 进行了修复。

```
inline void LinearTable::PushBackNode(RB_Node* node, const size_type& node_sz)
{
    FixSizeInc(back_node, node_sz); // 修复 Size 和 Sz
    back_node->right = node;
    node->parent = back_node;
    InsertFix(node); // 维护红黑树
    back_node = node;
}
```

代码中注释应该较为清晰，可以直接阅读实现过程，细节繁多复杂，就不再赘述。

- **前后节点**

  对于 front 和 back 由于我们维护了前后 node 所以可以很快的获得。

  ```
  inline value_type LinearTable::front()
  {
      return front_node->data.front();
  }

  inline value_type LinearTable::back()
  {
      return back_node->data.back();
  }
  ```

- **中间插入删除**

```
iterator insert(iterator, const value_type&); // 在 iterator 前 插入一个 val
iterator insert(iterator, size_type, const value_type&); // 在 iterator 前 插入 size 个 val

iterator erase(iterator); // 删除迭代器位置的元素
```

中间插入

单一插入的代码较长，在此不放出，直接谈谈，我的插入逻辑。

我的策略是插入大于一个节点的 BUF 后从前部弹出，不判断哪边近一些，因为逻辑已经十分复杂，再加逻辑代码将变得异常难以优化和 debug， 同时还可能得不偿失的产生负优化。

1. 首先我们先判断当前有无节点，如果无节点，那么我们需要处理，这边我直接调了 push_back
2. 如果有节点，该节点是否已经满了，如果没满直接插入，仅调整 sz
3. 如果该节点满了，那么找其前驱。这边又分两种情况
    - 如果没有左子树前驱，就不再找它的父前驱了，直接挂一个节点上去。
    - 如果有左子树前驱，这边又分两种情况，该前驱有没有满
        - 若该前驱没满，就直接插入
        - 如果满了，就在挂一个节点到其右节点上去

在这个过程中要同时注意以下两点

1. Size 和 Sz 的维护
2. FrontNode 的维护

可以看出这个逻辑是十分复杂的，代码实现起来也比较困难，在debug的过程中我也花费了相当长的时间。

中间删除

中间删除的过程更是阴间，在逻辑上并没有插入那么复杂，只是看一下当前迭代器所在节点是否删了之后就空了。

1. 如果没空就直接删，维护一下 sz;
2. 如果删了之后空了，就把它删掉，

在中间删除的过程中，该节点很可能是同时有两个子节点的，那么在二叉树删除维护上，我们一般采用将该节点和其后继调换，并删除其后继的方式来进行删除，难点在这个过程中要尤其注意 Size 和 Sz 的维护。

- **杂项**

```
void clear();
void swap(LinearTable&);

size_type size();
bool empty();
```

杂项包括以上四类函数，实现起来比较方便短小。

Clear

对于 clear 直接对树进行递归删除，直至最后剩一个nil节点，返回到默认初始化的结果，即 root, front, back == nil

```
inline void LinearTable::clear()
{
    DelTree(root);
    root = nil;
    back_node = root;
    front_node = root;
}

// 递归删除
void LinearTable::DelTree(RB_Node* TreeNode)
{
    if(TreeNode == nil)
        return;
    DelTree(TreeNode->left);
    DelTree(TreeNode->right);
    DelNode(TreeNode);
}
```

Swap

Swap 进行了两个线性表的交换，在内部就是指针的交换，比较方便

```
inline void LinearTable::swap(LinearTable& B)
{
    std::swap(root, B.root);
    std::swap(nil, B.nil);
    std::swap(back_node, B.back_node);
    std::swap(front_node, B.front_node);
}
```

Empty && Size

这两个是一起的，说到 Size 由于之前我们维护了 sz，所以可以直接在根节点 O(1) 获得，那么 empty 直接判断一下 sz 是否为 0 即可

```
inline size_type LinearTable::size()
{
    return root->sz;
}

inline bool LinearTable::empty()
{
    return root->sz == 0;
}
```

迭代器设计

基本信息

```
//
// ------------------------- Iterator --------------------------- //

class iterator : public
std::iterator<std::random_access_iterator_tag, value_type>
{
private:
    RB_Node* root;
    RB_Node* nil;
    key_type key;
    RB_Node* node_ptr;
    std::deque<value_type>::iterator inner_it;

    friend class LinearTable;
    friend difference_type operator-(const iterator& a, const iterator& b);
```

我的迭代器中储存了五个信息，一个是该表的 root 和 nil，还有该迭代器所指向的 key， 以及该迭代器所指向的外部红黑树节点 node_ptr 和内部 deque 的指针 inner_it。

支持方法

1. 前后自增自减
2. 加减运算
3. 迭代器减迭代器 (difference_type)
4. 加等减等
5. 比较 (等于，不等于)
6. 取值

迭代器失效问题

和 deque 一样我的迭代器在插入后也会产生失效问题。在插入和删除后，我会返回下一位置的迭代器。

性能测试

(单位: ms)

随机访问 - 1e7

复杂度对比： $O(1) <--> O(lg(\frac{n}{x}))$

调整SIZE	Deque	DequeTree
2^0	**97**	**3188**
2^1	**97**	**2436**
2^2	**97**	**2122**
2^3	**97**	**1891**
2^4	**97**	**1676**
2^5	**97**	**1564**
2^6	**97**	**1468**
2^7	**97**	**1350**
2^8	**97**	**1272**
2^9	**97**	**1207**

可以看到在随机访问方面，随着 NODE_BUF_SIZE 的调整可以获得更快的随机访问性能。

首尾插入 - 1e7

复杂度对比： $O(1) <--> O(lg(\frac{n}{x}))$

调整SIZE	Deque	DequeTree
2^0	2257	**33336**
2^1	2257	**19509**
2^2	2257	**11166**
2^3	2257	**7056**
2^4	2257	**4797**
2^5	2257	**4009**
2^6	2257	**3561**
2^7	2257	**3251**
2^8	2257	**3190**
2^9	2257	**2965**

在首尾插入上，随着 NODE_BUF_SIZE 的增大，节点数量减少，挂载节点次数变少，速度提升明显。

首尾删除 - 1e7

复杂度对比： $O(1) <--> O(lg(\frac{n}{x}))$

调整SIZE	Deque	DequeTree
2^0	1218	**34349**
2^1	1218	**23542**
2^2	1218	**10941**
2^3	1218	**6058**
2^4	1218	**3673**
2^5	1218	**2622**
2^6	1218	**2151**
2^7	1218	**1906**
2^8	1218	**1730**
2^9	1218	**1635**

首尾删除同理

中间插入 - Base 1e7, Insert 1e4

复杂度对比： $O(mn) <--> O(mlg(\frac{n}{x}) + mx)$

调整SIZE	Deque	DequeTree
2^3	12240	**135**
2^4	12240	**72**
2^5	12240	**46**
2^6	12240	**43**
2^9	12240	**36**
2^{12}	12240	**23**
2^{15}	12240	**49**
2^{18}	12240	**319**
2^{21}	12240	**2458**
2^{30}	12240	**12141**

对于中间插入我们可以看到时间呈现一个先慢后快的U形分布，在 NODE_BUF_SIZE == 2^{12} 次方的时候最快，这种情况的出现原因是由于在我的代码设计中，我在中间插入时不是每次都挂一个节点，而是寻找其前驱，如若前驱有空位就直接插入，这样可以大幅节省节点数量。那么在 1e4 的数据量下，对于小的 NODE_BUF_SIZE 其时间耗费不再体现在数据的移动上，而在于在树链上查找的前驱的时间，这时候节点数量越少，有利于快速插入。

中间批量插入 - Base 1e7, Insert 1e5

复杂度对比： $O(1) <--> O(lg([\frac{m}{x}]))$

调整SIZE	Deque	DequeTree
2^9	5	**51**
2^{10}	5	**44**
2^{11}	5	**56**
2^{12}	5	**75**
2^{13}	5	**45**
2^{14}	5	**117**
2^{15}	5	**294**
2^{16}	5	**620**
2^{17}	5	**1403**
2^{30}	5	**5**

对于批量的中间插入， deque可以预留出空间来，之后直接 fill，而相比之下树形结构虽然在该使用场景下不显现优势，但仍然不慢。可以注意到，我们的中间批量插入随着 size 的增长并不是稳定下降的，是先下降，再增长。当一个节点能够容纳数据量的时候直接掉成 deque 的复杂度。出现这个现象的原因比较玄学，在于 deque 的内存分配和我的树的内存分配耦合的问题，经过试验 2^9 即 512 相对来说是一个比较合适的 size

中间删除 - Base 1e7, erase 1e4

复杂度对比： $O(mn) <--> O(mlg(\frac{n}{x}) + mx)$

调整SIZE	Deque	DequeTree
2^4	460	**2.257**
2^7	460	**2.180**
2^8	460	**2.105**
2^9	460	**1.918**
2^{10}	460	**1.846**
2^{15}	460	**21.32**
2^{18}	460	**80.57**
2^{21}	460	**477.3**
2^{30}	460	**472.4**

可以看到中间删除的时间性能也是先随 SIZE 增大而减小，然后再次增大的。原因猜想应与中间插入类似。

OJ 测试

(以下测试均在默认 512 BUF_SIZE 下提交)

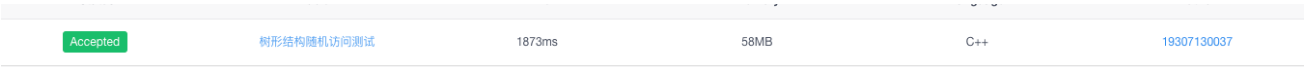
- 基本操作

- 批量插删

- 中间插值

可以发现在开启 O2 优化后，速度提升了很多，说明我的数据结构还有较大的提升空间，可以在代码方面进行进一步的常数优化。

- 树形随机访问



总结与展望

总结一下，这次数据结构的PJ，我采用了线性和树形结合的创新数据结构，在 deque 的首尾插入性能和红黑树的中间插入性能当中取得了一定的平衡，通过动态调整 NODE_BUF_SIZE 可以进一步动态的适应使用场景需求，是一个很有潜力的数据结构。

- **可能优化方向**
    1. 内存池
        - 可以通过内存池来对我的数据结构进行进一步优化，防止频繁的内存分配导致的性能下降。
    2. 线索化红黑树
        - 对每个节点可以多储存它的后继和前驱，通过线索化的方式进一步优化插入和访问性能。
    3. 动态 NODE_BUF_SIZE
        - 依照数据规模动态调整 NODE_BUF_SIZE 来自适应场景需求。

致谢

最后感谢孙未未老师和郭雨助教一学期的大力付出，这学期从一个零基础的数据结构算法小白成长到如今还算是入门级选手，多亏了老师和助教的辛勤付出。在寒假里我会继续优化我的数据结构，让它进一步获得更好的性能和表现，以及更多 fancy 的功能。

非常感谢！