

算法实验 3 实验报告

PB20000180 刘良宇


实验设备和环境

```
> neofetch

      .-/++00sssss00+/+-.
        `:+ssssssssssssssss+:`
          -+ssssssssssssssyyss+-
            .osssssssssssssdMMNNYssso.
              /ssssssssshdmmNnMmyNMmmHsssss/
                +sssssssshmYdMMMMMMMMddddyssssss+
                  /ssssssshNMMMMyhyyyyyhmNMMMNhsSSSSSS/
                    .sssssssdMMMNhsSSSSSSShNMMMdssssss.
                      +sssshhyNMMNySSSSSSSSSyNMMMySSSSSS+
                        ossyNMMMNyMMhsSSSSSSSSShmmhssssssso
                          ossyNMMMNyMMhsSSSSSSSSShmmhssssssso
                            +sssshhyNMMNySSSSSSSSSyNMMMySSSSSS+
                              .sssssssdMMMNhsSSSSSSShNMMMdssssss.
                                /ssssssshNMMMMyhyyyyhdNMMMNhsSSSSSS/
                                  +sssssssssdmYdMMMMMMMMddddyssssss+
                                    /ssssssssshdmNNNnMmyNMmmHhsSSSSS/
                                      .osssssssssssssssdMMMNysSSso.
                                        -+ssssssssssssssyyss+-
                                          `:+ssssssssssssss+:`
                                            .-/++00sssss00+/+-.

ubuntu@LAPTOP-EV8CNQ61

OS: Ubuntu 20.04.5 LTS on Windows 10 x86_64
Kernel: 5.15.79.1-microsoft-standard-WSL2
Uptime: 8 hours
Packages: 1354 (dpkg), 5 (snap)
Shell: zsh 5.8
Theme: Adwaita [GTK3]
Icons: Adwaita [GTK3]
Terminal: /dev/pts/8
CPU: AMD Ryzen 7 4800H with Radeon Graphics (16) @ 2.894GHz
GPU: 107d:00:00.0 Microsoft Corporation Device 008e
Memory: 1640MiB / 7626MiB
```



```
> g++ --version
g++ (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0
Copyright (C) 2019 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

实验内容及要求

实现区间树的基本算法，随机生成 30 个正整数区间，以这 30 个正整数区间的左端点作为关键字构建红黑树，先向一棵初始空的红黑树中依次插入 30 个节点，然后随机选择其中 3 个区间进行删除，最后对随机生成的 3 个区间 (其中一个区间取自 (25,30)) 进行搜索。实现区间树的插入、删除、遍历和查找算法。

实验方法和步骤

随机数据的生成

采用 `srand()`、`rand()` 和标准库的 `std::shuffle`。

前面两个函数主要用于随机区间的生成，最后一个函数用于生成随机区间后打乱，增强数据的随机性。

生成数据的代码：

```
srand(static_cast<unsigned>(time(nullptr)));
auto file = std::ofstream("./input/input.txt");
std::unordered_set<int> set;
std::vector<std::pair<int, int>> intervals;
// 要求：这里每行两个随机数据，表示区间的左右端点，至少 30 行
//      所有区间要么是 [0, 25] 的子区间，要么是 [30, 50] 的子区间
```

```

// 每个区间左端点互不相同
// 实现: 我们生成 15 个 [0, 25] 的子区间和 15 个 [30, 50] 的子区间
for (auto i = 0; i < 15; i++) {
    while (true) {
        const int int_low = rand() % 25;
        if (set.count(int_low) != 0U) {
            continue;
        }
        set.insert(int_low);
        const int int_high = rand() % (25 - int_low) + int_low;
        intervals.emplace_back(int_low, int_high);
        break;
    }
}
for (auto i = 0; i < 15; i++) {
    while (true) {
        const int int_low = rand() % 20 + 30;
        if (set.count(int_low) != 0U) {
            continue;
        }
        set.insert(int_low);
        const int int_high = rand() % (50 - int_low) + int_low;
        intervals.emplace_back(int_low, int_high);
        break;
    }
}
// 出于准确性考虑, 不妨先打乱数组
std::shuffle(intervals.begin(), intervals.end(),
             std::mt19937(std::random_device()()));
for (const auto &interval : intervals) {
    file << interval.first << ' ' << interval.second << std::endl;
}
file.close();

```

随机选择区间删除和查询的操作是类似的, 这里不再赘述。

区间树

尽可能采用了 modern cpp 的风格。

内存管理

通过使用智能指针, 我们可以避免手动的 `new` 和 `free` 内存管理。

具体设计上, 父节点指向子节点使用 `std::shared_ptr`, 子节点指向父节点用 `std::weak_ptr`。二者的区别是保存后者本身不会增加共享的智能指针的引用计数, 只有当通过 `lock()` 方法获取到一个智能指针实例后才会增加。这样一来就避免了 `std::shared_ptr` 循环引用引起的内存泄漏。

结点设计

```

struct Node {
    // 子结点使用 shared_ptr, 父结点使用 weak_ptr
    // 目的是为了保证自动的构造析构
    std::shared_ptr<Node> left;
    std::shared_ptr<Node> right;
    std::weak_ptr<Node> parent;
}

```

```

// 颜色定义
enum COLOR { BLACK = 0, RED };
COLOR color;
inline bool isRed() const { return color == RED; }
inline bool isBlack() const { return color == BLACK; }
inline void setRed() { color = RED; }
inline void setBlack() { color = BLACK; }
inline COLOR getColor() const { return color; }
inline void setColor(COLOR c) { color = c; }

// 存储区间
int int_low;
int int_high;
// 红黑树的附加信息
int max;

Node() = default;
Node(int int_low, int int_high)
    : int_low(int_low), int_high(int_high), max(int_high) {}
};

```

设计了一些实用的内联方法，颜色是枚举定义的。对红黑树的结点信息扩充：区间树结点需要额外保存 max 域。

区间树类

```

class IntervalTree {
    // 结点结构体
    struct Node {
        .....
    };

public:
    using shared = std::shared_ptr<Node>;
    using weak = std::weak_ptr<Node>;

private:
    // 哨兵结点
    shared nil = std::make_shared<Node>();
    // 根结点指针
    shared root = nil;

    // 其他的类方法
    .....
}

```

这里需要一个全局的空 nil 结点。初始根结点指向 nil 结点。

```

IntervalTree() {
    nil->setBlack();
    // 初始化 nil 结点的 max 域
    nil->max = std::numeric_limits<int>::min();
};

```

需要注意的是，在初始化区间树的时候，需要先将根节点的 max 域改为一个极小的负数，并且设置为黑色（为了保证满足红黑树的性质）。

维护 max 信息

区间树的大部分代码就是红黑树的代码，而红黑树的代码书上已经十分完备了，将伪代码用 C++ 写出来即可，因此下面主要介绍如何维护 max 信息这一书上没有介绍具体实现的部分。

1. 插入结点后更新信息
2. 插入结点后，rotate 时更新信息
3. 删除结点时更新信息
4. 删除结点后，rotate 时更新信息

第二点和第四点都是在 rotate 方法内实现：

```
void leftRotate(const shared &x) {
    ..... // 与红黑树代码相同
    // 维护附加信息
    y->max = x->max;
    x->max = std::max(std::max(x->int_high, x->left->max), x->right->max);
}

void rightRotate(const shared &y) {
    ..... // 与红黑树代码相同
    // 维护附加信息
    x->max = y->max;
    y->max = std::max(std::max(y->int_high, y->left->max), y->right->max);
}
```

这个更新是 $O(1)$ 可以完成的，因为可以明确，更上层的 max 域不会发生改变（底下还是这些结点）。具体调整方式就是代码里的两行。

第一点可以在插入完成后向上更新 parent，也可以插入时顺便更新沿途 parent 的 max 域：

```
void insertNode(shared z) {
    // insert
    weak y = nil;
    shared x = root;
    while (x != nil) {
        if (x->max < z->max) // 比较
            x->max = z->max; // 更新
        y = x;
        if (z->int_low < x->int_low)
            x = x->left;
        else
            x = x->right;
    }
    z->parent = y;
    // 后面是 z 作为 y 的孩子插入，同红黑树
    .....
}
```

第四点实际是通过更改 `transplant` 方法实现的，删除过程中只有这个方法会改变结点结构。

```
void transplant(const shared &u, const shared &v) {
    if (u->parent.lock() == nil) {
        root = v;
```

```

    } else if (u == u->parent.lock()->left) {
        u->parent.lock()->left = v;
    } else {
        u->parent.lock()->right = v;
    }
    v->parent = u->parent;
    // 维护新结点
    if (v != nil) {
        v->max =
            std::max(std::max(v->int_high, v->left->max), v->right->max);
    }
    // 从 v->parent 开始维护 max
    auto cur = v->parent.lock();
    while (cur != nil) {
        cur->max = std::max(std::max(cur->int_high, cur->left->max),
                           cur->right->max);
        cur = cur->parent.lock();
    }
}

```

需要注意的是，除了 parent 一直向上更新之外，新的结点本身也需要维护。

中序遍历

这是通过区间树类接收回调函数实现的。这里采用了 `std::function`，这样可以很方便的传入 lambda 函数。

```

private:
void traverseRecursive(
    const std::function<void(const IntervalTree::shared &)> &f,
    const shared &node, int depth) {
    if (node == nil) {
        return;
    }
    traverseRecursive(f, node->left, depth + 1);
    f(node);
    traverseRecursive(f, node->right, depth + 1);
}

public:
void traverse(const std::function<void(const IntervalTree::shared &)> &f) {
    traverseRecursive(f, root, 0);
}

```

main.cpp 里面就可以调用了。

```

// 再写入删除后的结果
// delete_file 是一个 ofstream
auto after_delete_traverse =
    [&delete_file](const IntervalTree::shared &node) {
        delete_file << node->int_low << ' ' << node->int_high << ' '
                    << node->max << std::endl;
    };
interval_tree.traverse(after_delete_traverse);

```

实验结果与分析

运行说明：项目目录使用 `g++ -O2 ./src/main.cpp -o main` 编译出可执行文件 main。

生成新的随机数据，并运行区间树：

```
./main -g
```

根据当前 input 数据运行区间树：

```
./main
```

运行	search.txt
<pre>> ./main 中序遍历结果已经保存在文件 删除结果已经保存在文件 搜索：28 28 搜索：27 29 搜索：43 46 搜索结果已经保存在文件</pre>	<pre>2-刘良宇-PB20000180-project 1 Null 2 Null 3 36 48 4</pre>

删除前后结果：

删除前	删除后
1 23 23	42 46
3 4 23	35 46
4 17 17	46 46
5 6 17	1 23 23
9 11 25	3 4 23
11 17 17	4 17 17
12 19 25	5 6 17
15 20 20	9 11 25
18 25 25	11 17 17
20 25 25	12 19 25
21 24 48	15 20 20
30 38 38	18 25 25
31 35 35	20 25 25
32 48 48	21 24 48
33 36 36	30 38 38
34 39 39	31 35 35
35 44 50	32 48 48
36 48 48	33 36 36
37 49 49	34 39 39
38 39 44	36 48 50
39 44 44	37 49 49
40 50 50	38 39 49
41 47 47	39 44 44
42 46 47	40 50 50
43 46 46	41 47 47
44 45 50	43 46 46
46 48 48	44 45 50
47 49 50	47 49 49
48 49 49	48 49 50
49 50 50	49 50 50

可以观察到的是，左端点为 42, 35, 46 的结点被移除了，并正确更新了各个结点的 max 域。