实验报告

杨凌林 PB17000083

实验设备与环境

实验设备

系统版本	Windows 10 专业版
系统版本号	2004
计算机型号	Asus U4100U
处理器	Core i7-8550U

编译环境

IDE	CLion(2020.2.4)
环境	MinGW
编译器	gcc(9.2.0)
CMake	Bundled(3.17.3)

实验内容和要求

实验一 红黑树

实现红黑树的基本算法， 分别对整数 $n = 20, 40, 60, 80, 100$，随机生成 n 个互异的正整数作为结点的关键字，向一棵初始为空的红黑树中依次插入 n 个节点，绘制**运行时间 - 问题规模**曲线，并将构建好的红黑树的中序遍历序列写入文件。

随机删除红黑树中 $n/4$ 个结点，统计删除操作所需时间，并绘制时间曲线图做分析。

实验二 区间树

实现区间树的基本算法，随机生成 30 个位于 $[0, 25]$ 或 $[30, 50]$ 的正整数区间，以这 30 个正整数区间的左端点作为关键字构建红黑树的节点，向一棵初始空的红黑树中依次插入这 30 个节点，然后随机选择其中 3 个区间进行删除。

实现区间树的插入、删除、遍历和查找算法，对随机生成的三个区间进行查找，判断是否与所创建的区间树有重叠的区间。

实验方法和步骤

实验一 红黑树

`src`文件夹中文件和函数的组织形式如下：

```
main.c
    int main()
RBT.h
    struct node   // 定义红黑树节点类型
    struct RBT    // 定义红黑树类型，包含两个 node 指针 root 和 nil
    struct RBT create()  // 创建一棵红黑树，并将其返回
    struct RBT LEFT_ROTATE(struct RBT T, struct node *x)  // 对红黑树节点 x 做左旋
    struct RBT RIGHT_ROTATE(struct RBT T, struct node *x) // 对红黑树节点 x 做右旋
    struct RBT RB_INSERT_FIXUP(struct RBT T, struct node *z)
        // 红黑树插入新节点后，沿节点 z 向上调整颜色
    struct RBT RB_INSERT(struct RBT T, struct node *z)
        // 向红黑树 T 插入新节点 z
    struct RBT insert(struct RBT T, int n)
        // 向红黑树插入关键字为 n 的新节点（调用 RB_INSERT 实现）
    struct node *MINIMUM(struct RBT T, struct node *z)
        // 返回红黑树 T 中以 z 为根节点子树中关键字最小的节点
    struct node *SUCCESSOR(struct RBT T, struct node *z)
        // 返回红黑树 T 中节点 z 的后继节点
    struct RBT RB_DELETE_FIXUP(struct RBT T, struct node *x)
        // 红黑树删除节点后，沿节点 x 向上调整颜色
    struct RBT RB_DELETE(struct RBT T, struct node *z)
        // 在红黑树 T 中删除节点 z
    struct node *find(struct RBT T, int n)
        // 返回红黑树 T 中关键字为 n 的节点指针，若没有，则返回 T.nil
    struct RBT delete(struct RBT T, int n)
        // 在红黑树 T 中删除关键字为 n 的节点
walk.h
    void print_color(struct node *x, FILE *f)
        // 将节点 x 的颜色写入文件 f
```

```
    void inorder_tree_walk(struct RBT T, struct node *x, FILE *f)
        // 对红黑树 T 的以 x 为根节点子树做中序遍历，遍历结果写入文件 f
rand_num_generate.h
    int *rand_num_generate(int n, int m)
        // 生成 n 个互不相同的正整数，范围在 1 - m
```

几点注意

- `main` 函数制作了一个小型菜单，通过读入用户输入执行相应操作。用户输入 `1` 执行生成随机数；输入 `2` 执行插入删除操作；输入 `0` 退出程序；输入其他数则程序提示用户应该输入正确范围内的数字。
- 为了方便对红黑树做操作，在实验中定义了红黑树的节点类型 `node` 和红黑树类型 `RBT` ，而对红黑树 `struct RBT T` 的引用，是通过 `T.root` 和 `T.nil` 两个 `node` 指针类型实现。
- 红黑树节点颜色 `color` 定义为 `int` 类型，`1` 代表当前节点为黑色，`0` 代表红色
- 在中序遍历时，为方便检查，连同节点颜色、`size` 一并输出。

实验二 区间树

`src` 文件夹中文件和函数的组织形式如下：

```
main.c
    int main()
interval_tree.h
    int three_num_max(int a, int b, int c) //返回 a, b, c 三数中最大值
    struct node  // 区间树节点类型
    struct IT    // 区间树类型，包含两个 node 指针 root 和 nil
    struct IT create()    // 创建一棵区间树，并将其返回
    struct IT LEFT_ROTATE(struct IT T, struct node *x)   // 对区间树节点 x 做左旋
    struct IT RIGHT_ROTATE(struct IT T, struct node *x)  // 对区间树节点 x 做右旋
    struct IT IT_INSERT_FIXUP(struct IT T, struct node *z)
        // 区间树插入新节点后，沿节点 z 向上调整颜色
    struct IT IT_INSERT(struct IT T, struct node *z)
         // 向区间树 T 插入新节点 z
    struct IT insert(struct IT T, int n1, int n2)
        // 向区间树插入区间 [n1, n2] 的新节点 (调用 IT_INSERT 实现)
    struct node *MINIMUM(struct IT T, struct node *z)
        // 返回区间树 T 中以 z 为根节点子树中关键字最小的节点
    struct node *SUCCESSOR(struct IT T, struct node *z)
        // 返回区间树 T 中节点 z 的后继节点
    struct IT IT_DELETE_FIXUP(struct IT T, struct node *x)
        // 区间树删除节点后，沿节点 x 向上调整颜色
    struct IT IT_DELETE(struct IT T, struct node *z)
        // 在区间树 T 中删除节点 z
    struct node *find(struct IT T, int n1, int n2)
        // 返回区间树 T 中区间为 [n1, n2] 的节点指针，若没有，则返回 T.nil
    struct IT delete(struct IT T, int n1, int n2)
        // 在区间树 T 中删除区间为 [n1, n2] 的节点
    int overlap_or_not(struct node *x, int n1, int n2)
        // 判断 x 区间是否和 [n1, n2] 重叠，有重叠返回 1，否则返回 0
    struct node *IT_search(struct IT T, int n1, int n2)
        // 返回与区间 [n1, n2] 有重叠的区间，若不存在，返回 T.nil
    void search(struct IT T, int n1, int n2, FILE *f)
        // 将与区间 [n1, n2] 有重叠的区间写入文件 f，若不存在，写入 NULL
walk.h
```

```
    void inorder_tree_walk(struct IT T, struct node *x, FILE *f)
        // 对区间树 T 的以 x 为根节点子树做中序遍历，遍历结果写入文件 f
rand_int_generate.h
    int *rand_index_generate(int range, int n)
        // 从 [0, range-1] 中不重复生成 n 个整数
    int *rand_intnum_generate(int n)
        // 从实验要求中给定的区间范围内随机生成左端点不重复的 n 个区间
    void rand_int_generate(FILE *f)
        // 从实验要求中给定的区间范围内随机生成左端点不重复的 30 个区间，写入文件 f (
调用rand_intnum_generate)
    void rand_int_for_search_generate(int *data_search_low, int
*data_search_high)
        // 生成三个区间，存储在 data_search_low, data_search_high 中
        // 第一个区间位于 (25, 30) (i.e. [26, 29]) 中
        // 第二个区间位于 [0, 25] 中
        // 第三个区间位于 [30, 50] 中
```

几点注意

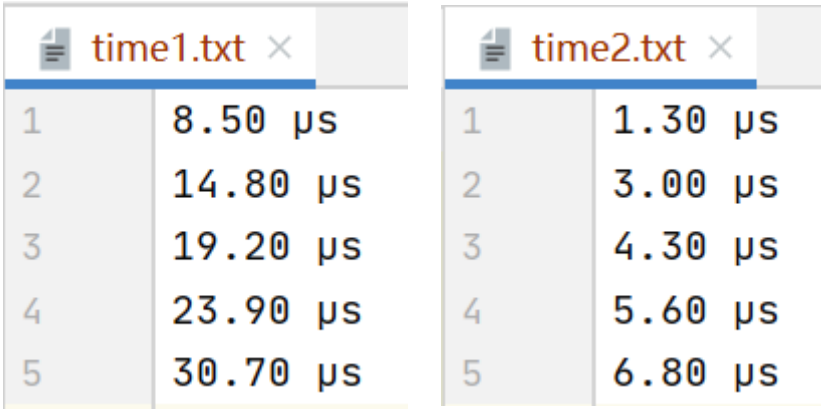
- 为了方便对区间树做操作，在实验中定义了区间树的节点类型 `node` 和区间树类型 `IT`，而对区间树 `struct IT T` 的引用，是通过 `T.root` 和 `T.nil` 两个 `node` 指针类型实现。
- 区间树节点关键字为区间左端点 `low`；颜色 `color` 定义为 `int` 类型，`1` 代表当前节点为黑色，`0` 代表红色；`max` 为以该节点为根的子树中最大的右端点的值。

实验结果和分析

实验一 红黑树

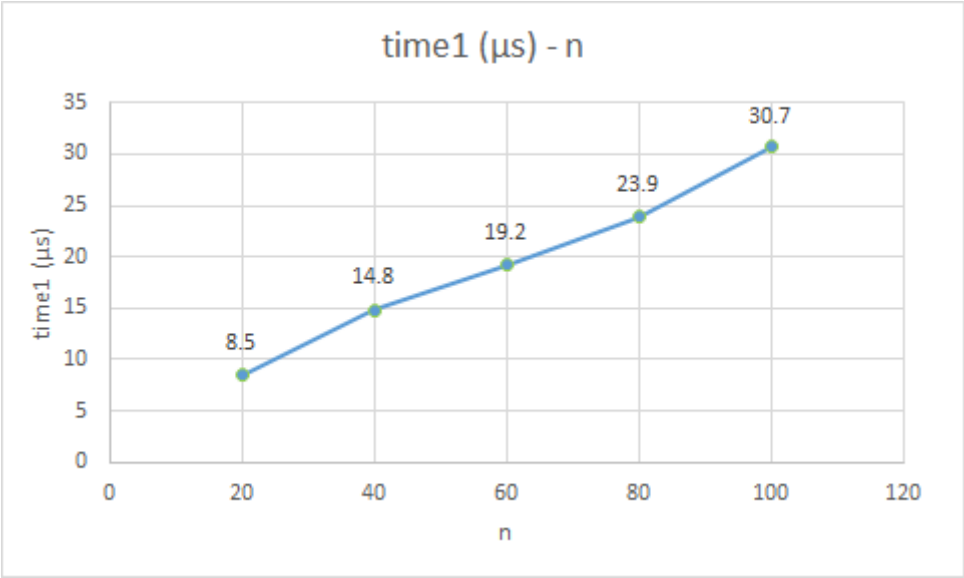
实验截图

红黑树插入操作运行时间 `time1.txt` 和删除操作运行时间 `time2.txt` 文件结果截图如下：

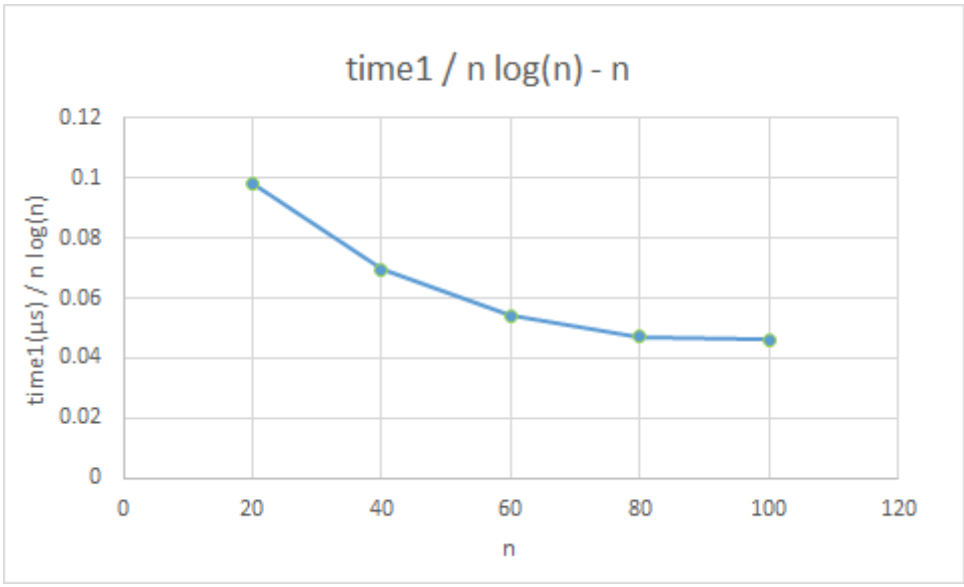


实验结果分析

先分析插入操作，对实验运行时间做图（`time1 - n` 图）：

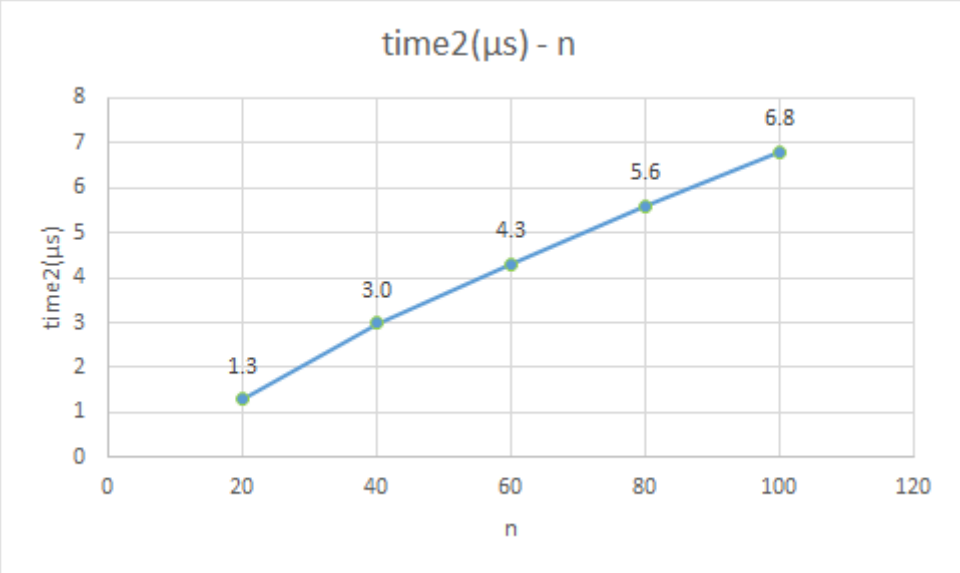


初看图像，可能会觉得 `time1 - n` 是线性关系（理论复杂度为 $\mathcal{O}(n\log(n))$，因为一次插入操作复杂度为 $\mathcal{O}(\log m)$，m 为红黑树中节点个数，一共插入 n 个节点，复杂度为 $\mathcal{O}(n\log(n))$），但我们仍需做进一步分析。我们可以绘制 `time1/nlog(n) - n` 图，如下所示：

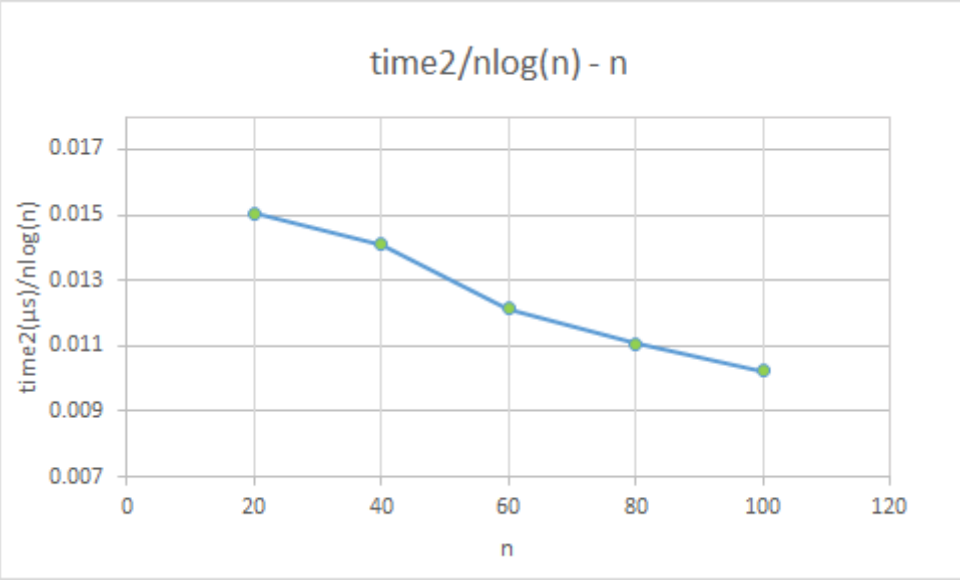


可以看出，随着 n 的增大，`time1/nlog(n)` 趋于平缓，且收敛的速度很快，故我们可以判断插入操作的复杂度为 $\mathcal{O}(n\log(n))$，即插入一个节点操作的复杂度为 $\mathcal{O}(\log(n))$，这与理论复杂度是相吻合的。

再看删除操作，同样对实验运行时间做图（`time2 - n` 图）：



而删除操作的理论复杂度为 $\mathcal{O}(\log(n))$（删除一个节点理论复杂度为 $\mathcal{O}\log(m)$，m 为红黑树当前节点个数，实验中共删除 $n/4$ 个节点，理论复杂度为 $\mathcal{O}(n\log(n))$。绘制 `time2/nlog(n) - n` 图，如下所示：



实验开始阶段，数据量较小，随着数据量 n 增大，可以看到 `time2/nlog(n)` 趋于平缓，这就验证了理论复杂度 $\mathcal{O}(n\log(n))$ 是正确的，即删除一个节点的复杂度为 $\mathcal{O}(\log(n))$。

实验二 区间树

实验截图

实验结果均已输出到 `output` 文件夹中对应文件，这里仅对查找操作做展示：

search.txt

```
Three intervals to search:
26 27
6 7
36 42

NULL
4 15
37 47
```

可以看到程序能够正确运行给出相应结果。

实验结果分析

通过添加节点的 `max` 值，我们很容易对区间重叠查找问题得到性能较好的算法，而复杂度不会产生量级上的增大。

实验总结

通过对实验一的运行时间作图，我们可以验证红黑树节点插入删除的时间复杂度是正确的。

我们在实验一中，通过红黑树的构造，能够得到平衡度较好的二叉检索树；而实验二中，通过添加节点的 `max` 值，对区间查找问题我们得到了性能较好的算法。

我们在以后的学习中应该合理运用数据结构的扩张，对于特定问题，添加合适的属性，能够简化对问题的处理，同时不会增大原数据结构的复杂度。