电子科技大学

计算机专业类课程

实验报告

课程名称：数据结构与算法

学院专业：计算机科学与工程学院（互联网+）

学生姓名：李逢君

学　　号：**2016060601010**

指导教师：周益民

日　　期：**2018** 年 **10** 月 **10** 日

实验报告撰写说明

此标准实验报告模板系周益民发布。其中含有三份完整实验报告的示例。

黑色文字部分，实验报告严格保留。公式为黑色不在此列。
蓝色文字部分，需要按照每位同学的理解就行改写。
红色文字部分，删除后按照每位同学实际情况写作。

在实验素材完备的情况下，报告写作锻炼写作能力。实验最终的评定成绩与报告撰写的工整性、完整性密切相关。也就是，你需要细节阐述在实验中遇到的问题，解决的方案，多样性的测试和对比结果的呈现。图文并茂、格式统一。

电 子 科 技 大 学

实 验 报 告

实验一

一、实验名称：

二叉树的应用：二叉排序树 BST 和平衡二叉树 AVL 的构造

二、实验学时：4

三、实验内容和目的：

树型结构是一类重要的非线性数据结构。其中以树和二叉树最为常用，直观看来，树是以分支关系定义的层次结构。树结构在客观世界中广泛存在，如人类社会的族谱和各种社会组织机构都可用树来形象表示。树在计算机领域中也得到广泛应用，如在编译程序中，可用树来表示源程序的语法结构。又如在数据库系统中，树型结构也是信息的重要组织形式之一。

实验内容包含有二:

二叉排序树(Binary Sort Tree)又称二叉查找(搜索)树(Binary Search Tree)。其定义为：二叉排序树或者是空树，或者是满足如下性质的二叉树：1.若它的左子树非空，则左子树上所有结点的值均小于根结点的值；2.若它的右子树非空，则右子树上所有结点的值均大于根结点的值；3.左、右子树本身又各是一棵二叉排序树。

平衡二叉树（Balanced Binary Tree）又被称为 AVL 树。具有以下性质：它是一棵空树或它的左右两个子树的高度差的绝对值不超过 1，并且左右两个子树都是一棵平衡二叉树。构造与调整方法。

实验目的:

二叉排序树的实现

(1) 用二叉链表作存储结构，生成一棵二叉排序树 T。

(2) 对二叉排序树 T 作中序遍历，输出结果。

(3) 输入元素 x,查找二叉排序树 T,若存在含 x 的结点,则返回结点指针。

平衡二叉树的实现

(1) 用二叉链表作为存储结构，输入数列 L，生成一棵平衡二叉树 T。

(2) 对平衡二叉树 T 作中序遍历，输出结果。

四、实验原理

二叉排序树的实现

(1) 二叉排序树节点的插入

在二叉排序树中插入新节点，要保证插入后的二叉树仍然符合二叉排序树的定义，即二叉排序树中的每一个节点的左子树的值都要比该节点小，右子树的值都要比该节点大。插入过程：若二叉排序树为空，则待插入节点*Node 作为根节点插入到空树中；若二叉排序树非空，将待插入节点的值*Node->data 与根节点的值*TreeRoot->data 进行比较，若*Node->data=*TreeRoot->data，则无需插入；若 *Node->data<*TreeRoot->data，则该值插入到根节点的左子树中；若 *Node->data>*TreeRoot->data，则该值插入到根节点的右子树中。在子树的插入过程和在树中的插入过程相同，如此进行下去，直到把数列的每一个节点*Node 作为一个新的叶子插入到二叉排序树中，或者直到发现树已有相同的值的为止。

(2) 生成了一棵二叉排序树

1. 每次插入的新结点都是二叉排序树上新的叶子结点。

2. 由不同顺序的关键字序列，会得到不同二叉排序树。

3. 对于一个任意的关键字序列构造一棵二叉排序树，其实质上对关键字进行排序。

(3) 二叉排序树节点的删除

在二叉排序树中删除节点,要保证删除后的二叉树仍然符合二叉排序树的定义，即二叉排序树中的每一个节点的左子树的值都要比该节点小，右子树的值都要比该节点大。要删除二叉排序树中的某一个节点首先需要查找该节点在二叉排序树中是否存在，将待删除节点的值*Node->data 与二叉排序树的根节点的值 *TreeRoot->data 进行比较，若*Node->data=*TreeRoot->data,则证明查找到待删除节点，进行下一步删除操作；若*Node->data<*TreeRoot->data，则将待删除节点与根节点的左孩子的值进行比较；若*Node->data<*TreeRoot->data，则将待删除节点与根节点的右孩子的值进行比较；直到查找到待删除节点为止，或者二叉排序树不存在该节点，终止删除操作。删除过程要分三种情况：①待删除节点为二叉排序树的叶子节点，删除该节点不会影响二叉排序树的特性，只需将其父节点

相应的指针域改为空指针即可；②待删除节点为只含有一棵子树的节点，删除该节点只需令其父节点相应的指针域指向该节点对应的子树即可；③待删除节点为含有左右子树的节点，删除该节点需要使用其中序遍历的前驱代替之，代替之后将其前驱删除。换言之，即使用待删除节点右子树中的最小值代替该节点的值，并将最小值所处的节点删除。

平衡二叉树的实现

（1）平衡二叉树节点的插入

1. 使用上述构建二叉排序树的方法对节点进行插入。

2. 因为二叉排序树的插入是将新节点插入到树的叶子节点，因此可以在插入递归返回的过程中更新每个节点的平衡因子*Node->bf，即计算该节点的左子树与右子树高度的差。

3. 根据 A、B 的平衡因子，判断是否失衡及失衡类型，并作旋转处理。

4. 如果平衡因子*Node->bf>=2，则证明节点在左子树插入导致失衡，此时分两种情况，①插入的值小于失衡节点左子树的值即 data < (*Node)->lchild->data，则证明失衡是在该节点的左子树的左子树插入节点导致的，进行 LL 型平衡旋转；②插入的值大于失衡节点左子树的值即 data > (*Node)->lchild->data，则证明失衡是在该节点的左子树的右子树插入节点导致的，进行 LR 型平衡旋转。

5. 如果平衡因子*Node->bf<=-2，则证明节点在右子树插入导致失衡，此时分两种情况，①插入的值大于失衡节点右子树的值即 data>(*Node)->rchild->data，则证明失衡是在该节点的右子树的右子树插入节点导致的，进行 RR 型平衡旋转；②插入的值小于失衡节点左子树的值即 data<(*Node)->lchild->data，则证明失衡是在该节点的右子树的左子树插入节点导致的，进行 RL 型平衡旋转。

LL 型平衡旋转

由于在 A 的左孩子的左子树上插入结点使 A 的平衡因子由 1 增到 2 而使树失去平衡。调整方法是将子树 A 进行一次顺时针旋转。具体方法是将失衡节点的左指针指向其左孩子的右子树，将其左孩子的右指针指向该失衡节点。随即更新这两个节点的平衡因子。

RR 型平衡旋转

其实这和第一种 LL 型是镜像对称的，由于在 A 的右子树的右子树上插入结点使得 A 的平衡因子由 1 增为 2；解决方法也类似，只要进行一次逆时针旋转。具体方式是将失衡节点的右指针指向其右孩子的左子树，将其右孩子的左指针指向该失衡节点。随即更新这两个节点的平衡因子。

LR 型平衡旋转

这种情况稍复杂些。是在 A 的左子树的右子树 C 上插入了结点引起失衡。但具体是在插入在 C 的左子树还是右子树却并不影响解决方法，只要进行两次旋转（先逆时针，后顺时针）即可恢复平衡。具体操作是先对失衡节点的左孩子进行左旋（RR）操作，然后再对失衡节点本身进行右旋（LL）操作。

RL 平衡旋转

也是 LR 型的镜像对称，是 A 的右子树的左子树上插入的结点所致，也需进行两次旋转(先顺时针，后逆时针)恢复平衡。具体操作是先对失衡节点的右孩子进行右旋（LL）操作，然后再对失衡节点本身进行左旋（RR）操作。

(2) 生成了一棵平衡二叉树

1. 每次插入的新结点都是平衡二叉树上新的叶子结点。

2. 由于插入节点可能会破坏平衡二叉树的性质，因此在插入节点后需要进行调整以维护二叉树的平衡性。

3. 由不同顺序的关键字序列，可能会得到不同二叉排序树。平衡二叉树既持了二叉排序树排序的性质，又降低了原来操作的复杂度。

(3) 平衡二叉树的删除

在二叉排序树中删除节点,要保证删除后的二叉树仍然符合平衡二叉树的定义。对于排序性的维护，可以延用上述二叉排序树的做法。对于平衡性的维护，可以借用平衡二叉树生成的逆向思维。

若删除节点导致以该节点位置为根的平衡二叉树的平衡因子大于等于 2，可以看做往该平衡二叉树的左子树插入节点导致的不平衡,①若删除节点的值大于等于根的左孩子即的值 data>= (*TreeRoot)->lchild->data，说明删除是在根的左孩子的及其右子树进行的，可以看成向根的左孩子的左子树增加节点导致的不平衡，进行 LL 旋转。②若删除节点的值小于根的左孩子即 data< (*TreeRoot)->lchild->data，则说明删除是在根的左孩子的左子树进行的，可以看成向根的左孩子的右子树增加节点导致的不平衡，进行 LR 旋转。

若删除节点导致以该节点位置为根的平衡二叉树的平衡因子小于等于-2，可以看做往该平衡二叉树的右子树插入节点导致的不平衡,①若删除节点的值小于等于根的右孩子的值即 data <= (*TreeRoot)->rchild->data，说明删除是在根的右孩子的及其左子树进行的,可以看成向根的右孩子的右子树增加节点导致的不平衡，进行 RR 旋转。②若删除节点的值大于根的右孩子即 data > (*TreeRoot)->rchild->data，则说明删除是在根的右孩子的右子树进行的，可以看成向根的右孩子的左子树增加节点导致的不平衡，进行 RL 旋转。

五、实验器材（设备、元器件）

处理器：Intel® Core™ i7-6700HQ CPU @ 2.60GHz　2.60GHz

已安装的内存(RAM)：8GB

系统类型：64 位操作系统，基于 x64 的处理器

IDE：CLion 2018.2.4

Environment：mingw32 (Version: w64 6.0)

cMake：Bundled (Version: 3.12.3)

Debugger: MinGW-w64 GDB (Version: 8.1)

六、实验步骤

二叉排序树，二叉平衡树

```
1.  ******************************二叉排序树********************************
2.  typedef int DataType;
3.  // 二叉排序树结构定义
4.  typedef struct BSTNode {
5.      DataType data;
6.      struct BSTNode *lchild, *rchild;
7.  } BSTree, *BSTreeP;
8.  #define MaxSize 100;
9.
10. ********************************相关操作*********************************
11. // 初始化一个二叉排序树节点
12. BSTreeP InitNode(DataType data) {
13.     BSTreeP NodeP = (BSTreeP) malloc(sizeof(BSTree)); // 分配空间
14.     NodeP->data = data;  // 赋值
15.     NodeP->lchild = NodeP->rchild = NULL; // 左右指针为 NULL
16.     return NodeP;
17. }
18.
19. // 二叉排序树插入
20. void BSTreeInsertNode(BSTreeP *TreeRoot, DataType data) {
21.     // 新建一个节点
22.     BSTreeP NodeP = InitNode(data);
23.     // 如果树根还没有创建则将该新建的节点赋值给根
24.     if (*TreeRoot == NULL) {
25.         *TreeRoot = NodeP;
26.     } // 新插入的节点数值比根小或等，走根的左边
27.     else if (data <= (*TreeRoot)->data) {
28.         if ((*TreeRoot)->lchild == NULL) { // 左子树递归结束处
29.             (*TreeRoot)->lchild = NodeP;  // 插入到左孩子
30.         } else { // 左递归
31.             BSTreeInsertNode(&((*TreeRoot)->lchild), data);
32.         }
33.     } else { // 新插入的节点数值比根大，走根的右边
34.         if ((*TreeRoot)->rchild == NULL) { // 右子树递归结束处
35.             (*TreeRoot)->rchild = NodeP;  // 插入到右孩子
36.         } else { // 右递归
37.             BSTreeInsertNode(&((*TreeRoot)->rchild), data);
38.         }
39.     }
40. }
41.
42. // 搜索排序二叉树
43. int SearchBST(BSTreeP TreeRoot, int value, BSTreeP *Node, BSTreeP *ParentNode) {
44.     // 在根指针 bt 所指二叉排序树中递归地查找关键字等于 key 的结点，若查找成功，指针 Node 指向该结点，并返回 1；否则指针 Node 指向查找路径最后一个
45.     // 访问结点并返回 0,指针 ParentNode 指向 bt 的双亲，初始调用时值为 NULL
46.     if (TreeRoot == NULL) { // 查找失败
47.         *Node = *ParentNode;
48.         return 0;
49.     } else if (value == TreeRoot->data) { // 查找成功
50.         *Node = TreeRoot;
51.         return 1;
```

```
52.     } else if (value < TreeRoot->data) { // 左子树继续查找
53.         *ParentNode = TreeRoot;
54.         return SearchBST(TreeRoot->lchild, value, Node, ParentNode);
55.     } else { // 右子树继续查找
56.         *ParentNode = TreeRoot;
57.         return SearchBST(TreeRoot->rchild, value, Node, ParentNode);
58.     }
59. }
60.
61. // 删除某节点，先搜索，后删除
62. void DeleteBST(BSTreeP *TreeRoot, int value) {
63.     BSTreeP Node = NULL, ParentNode = NULL;
64.     if (SearchBST(*TreeRoot, value, &Node, &ParentNode) == 0) { // 对该节点进行搜索，如果没有则返回，若有则继续
65.         printf("fail, not exit\n");
66.         return;
67.     } else {
68.         if (Node->lchild == NULL && Node->rchild == NULL) {     // 判断为叶子节点
69.             if (ParentNode == NULL) {   // 如果有且只有一个根节点
70.                 free(Node);     //  释放该节点
71.             } else if (ParentNode->lchild == Node) {    // 如果该叶子节点是左孩子
72.                 ParentNode->lchild = NULL;  // 其父节点的右孩子指向 NULL
73.             } else {    // 如果该叶子节点是右孩子
74.                 ParentNode->rchild = NULL;  //  其父节点的右孩子指向 NULL
75.             }
76.             free(Node);     //  释放待删除节点
77.         } else if (Node->lchild == NULL && Node->rchild != NULL) { // 判断只有右子树
78.             if (ParentNode == NULL) {   //  如果父节点为根节点
79.                 *TreeRoot = Node->rchild;   // 树根指针指向其右子树
80.             } else if (ParentNode->lchild == Node) {    // 如果为父节点的左子树
81.                 ParentNode->lchild = Node->rchild;  //  父节点的左指针指向该节点的左孩子
82.             } else {    // 如果为父节点的右子树
83.                 ParentNode->rchild = Node->rchild;  //  父节点的右指针指向该节点的右孩子
84.             }
85.             free(Node);     //  释放待删除节点
86.         } else if (Node->rchild == NULL && Node->lchild != NULL) { // 判断只有左子树
87.             if (ParentNode == NULL) {   //  如果父节点为根节点
88.                 *TreeRoot = Node->lchild;   // 树根指针指向其左子树
89.             } else if (ParentNode->lchild == Node) {    // 如果为父节点的左子树
90.                 ParentNode->lchild = Node->lchild;  //  父节点的左指针指向该节点的左孩子
91.             } else {    // 如果为父节点的右子树
92.                 ParentNode->rchild = Node->lchild;  //  父节点的右指针指向该节点的右孩子
93.             }
94.             free(Node);     //  释放待删除节点
95.         } else { // 判断含有左右子树
96.             BSTreeP minParentNode = Node;
97.             // 查找右子树的最小值，来替换它
98.             BSTreeP minNode = Node->rchild;
99.             while (minNode->lchild != NULL) { // 判断是否为最小值，若是则停止，否则继续迭代查找最小值
100.                minParentNode = minNode;    //  记录最小值节点的父节点
101.                minNode = minNode->lchild;  //  记录最小值节点
102.            }
103.            Node->data = minNode->data;
104.            if (minParentNode != Node) { // 进了 while，最小值节点为待删除节点右子树的最左节点
105.                minParentNode->lchild = minNode->rchild;    // 最小值父节点的左指针指向最小值节点的右子树
106.            } else {    // 没进 while，最小值节点为待删除节点的右节点
107.                minParentNode->rchild = minNode->rchild;    // 最小值父节点的右指针指向最小值节点的右子树
108.            }
109.            free(minNode);  //  释放待删除节点
110.        }
111.    }
112. }
113.
114. ********************************各种遍历*********************************
115. // 前序遍历
116. void PreOrder(BSTreeP TreeRoot) {
117.     if (TreeRoot != NULL) {
118.         NodeVisit(TreeRoot->data);
119.         PreOrder(TreeRoot->lchild);
120.         PreOrder(TreeRoot->rchild);
121.     }
122. }
123.
124. // 中序遍历
125. void InOrder(BSTreeP TreeRoot) {
126.     if (TreeRoot != NULL) {
127.         InOrder(TreeRoot->lchild);
128.         NodeVisit(TreeRoot->data);
129.         InOrder(TreeRoot->rchild);
130.     }
131. }
132.
133. // 层次遍历
134. void LevelOrder(BSTreeP TreeRoot) {
```

```
    BSTreeP Queue[MaxSize]; // 定义队列
    int front = -1, rear = 0;
    if (TreeRoot == NULL) return; // 空二叉树，遍历结束
    Queue[rear] = TreeRoot; // 根结点入队列
    while (rear != front) { // 若队列不空，继续遍历。否则，遍历结束
        front++; // 出队
        NodeVisit(Queue[front]->data); // 访问刚出队的元素
        if (Queue[front]->lchild != NULL) { // 如果有左孩子，左孩子入队
            rear++;
            Queue[rear] = Queue[front]->lchild;
        }
        if (Queue[front]->rchild != NULL) { // 如果有右孩子，右孩子入队
            rear++;
            Queue[rear] = Queue[front]->rchild;
        }
    }
}

// 打印值
void NodeVisit(DataType data) {
    printf("%d ", data);
}

*******************************可视化部分*******************************
// 将二叉树生成 dot 文件并生成 png
void createDotFile(const char *filename, BSTreeP TreeRoot) {
    FILE *fp = fopen(filename, "w");    // 文件指针
    if (fp == NULL) {   // 为 NULL 则返回
        printf("File cannot open!");
        exit(0);
    }
    fprintf(fp, "digraph G {\n");   // 开头
    // 利用层次遍历构造
    BSTreeP Queue[MaxSize]; // 定义队列
    int front = -1, rear = 0;
    if (TreeRoot == NULL) return; // 空二叉树，遍历结束
    Queue[rear] = TreeRoot; // 根结点入队列
    while (rear != front) { // 若队列不空，继续遍历。否则，遍历结束
        front++; // 出队
        NodeVisit(Queue[front]->data); // 访问刚出队的元素
        fprintf(fp, "%d [shape=circle];\n", Queue[front]->data);
        if (Queue[front]->lchild != NULL) { // 如果有左孩子，左孩子入队
            rear++;
            Queue[rear] = Queue[front]->lchild;
            fprintf(fp, "%d->%d;\n", Queue[front]->data, Queue[front]->lchild->data);
        }
        if (Queue[front]->rchild != NULL) { // 如果有右孩子，右孩子入队
            rear++;
            Queue[rear] = Queue[front]->rchild;
            fprintf(fp, "%d->%d;\n", Queue[front]->data, Queue[front]->rchild->data);
        }
    }
    fprintf(fp, "}\n"); // 结尾
    fclose(fp); // 关闭 IO
}

*******************************作业算法设计*******************************
int Depth(BSTreeP TreeRoot);    //  计算树的深度
int CountLeaf(BSTreeP TreeRoot);    // 计算叶子的个数
int CountBiNode(BSTreeP TreeRoot);  // 计算总节点个数
void Exchange(BSTreeP TreeRoot);    // 交换左右子树
void CBTreeInsertNode(BSTreeP *TreeRoot, DataType value); // 创建一个完全二叉树

*******************************平衡二叉树*******************************
typedef int DataType;
// 平衡二叉树结构定义
typedef struct AVLNode {
    DataType data;
    int bf; // 平衡因子
    struct AVLNode *lchild, *rchild;
} AVLTree, *AVLTreeP;
#define MaxSize 100;

*******************************相关操作*******************************
// 初始一个空节点
AVLTreeP InitNode(DataType value) {
    AVLTreeP Node = (AVLTreeP) malloc(sizeof(AVLTree)); // 分配内存
    Node->data = value; // 赋值
    Node->bf = 0;   // 初始平衡因子为 0
    Node->lchild = Node->rchild = NULL; //  左右指针指向 NULL
    return Node;
}

// 平衡二叉树插入节点
void AVLInsertNode(AVLTreeP *TreeRoot, DataType data) {
    // 新建一个节点
```

```
    AVLTreeP NodeP = InitNode(data);
    // 如果树根还没有创建则将该新建的节点赋值给根
    if (*TreeRoot == NULL) {
        *TreeRoot = NodeP;
    } else if (data <= (*TreeRoot)->data) { // 新插入的节点数值比根小或等，走根的左边
        // 左子树递归结束处
        if ((*TreeRoot)->lchild == NULL) {
            (*TreeRoot)->lchild = NodeP;
        } else { // 左递归
            AVLInsertNode((&(*(TreeRoot))->lchild), data);
            AVLTreeFix(&(*TreeRoot), data, 1);
        }
    } else {
        // 右子树递归结束处
        if ((*TreeRoot)->rchild == NULL) {
            (*TreeRoot)->rchild = NodeP;
        } else { // 右递归
            AVLInsertNode((&(*TreeRoot)->rchild), data);
            AVLTreeFix(&(*TreeRoot), data, 1);
        }
    }
    (*TreeRoot)->bf = updateBF(&(*TreeRoot)); // 树高等于子树高度加一
}

// AVL 删除节点，参考自 https://blog.csdn.net/pjmike233/article/details/81545136
AVLTreeP DeleteAVL(AVLTreeP *TreeRoot, int data) {
    if (TreeRoot == NULL) {
        return NULL;
    }
    if (data < (*TreeRoot)->data) { // 左递归
        (*TreeRoot)->lchild = DeleteAVL(&((*TreeRoot)->lchild), data);
    } else if (data > (*TreeRoot)->data) {  // 右递归
        (*TreeRoot)->rchild = DeleteAVL(&((*TreeRoot)->rchild), data);
    } else {  // 找到待删除的节点
        if ((*TreeRoot)->lchild != NULL && (*TreeRoot)->rchild != NULL) {  // 删除的节点有两个孩子
            (*TreeRoot)->data = getRchildMin(&((*TreeRoot)->rchild));   // 找到右子树的最小值节点并替换该值
            (*TreeRoot)->rchild = DeleteAVL(&((*TreeRoot)->rchild), (*TreeRoot)->data);   // 删除右子树的最小值节点
        } else {    // 删除的节点只有一个孩子或者没有孩子
            AVLTreeP temp = *(TreeRoot);
            (*TreeRoot) = ((*TreeRoot)->lchild != NULL) ? (*TreeRoot)->lchild : (*TreeRoot)->rchild;  // 判断删除是否为左孩子，若是则赋给左孩子，否则赋给右孩子
            free(temp); // 释放空间
        }
    }
    // 恢复二叉树的平衡
    if (*TreeRoot == NULL) {    //  如果根为 NULL，则返回 NULL
        return *TreeRoot;
    }
    (*TreeRoot)->bf = updateBF(&(*TreeRoot));    // 更新权重
    AVLTreeFix(&(*TreeRoot), data, 0);  //  恢复二叉树的平衡性
    return *TreeRoot;
}

// 搜索 AVL 树
int SearchAVL(AVLTreeP TreeRoot, int value, AVLTreeP *Node, AVLTreeP *ParentNode) {
    // 在根指针 bt 所指二叉排序树中递归地查找关键字等于 key 的结点，若查找成功，指针 Node 指向该结点，并返回 1；否则指针 Node 指向查找路径最后一个
    // 访问结点并返回 0,指针 ParentNode 指向 bt 的双亲，初始调用时值为 NULL
    if (TreeRoot == NULL) { // 查找失败
        *Node = *ParentNode;
        return 0;
    } else if (value == TreeRoot->data) { // 查找成功
        *Node = TreeRoot;
        return 1;
    } else if (value < TreeRoot->data) { // 左子树继续查找
        *ParentNode = TreeRoot;
        return SearchAVL(TreeRoot->lchild, value, Node, ParentNode);
    } else { // 右子树继续查找
        *ParentNode = TreeRoot;
        return SearchAVL(TreeRoot->rchild, value, Node, ParentNode);
    }
}

********************************各种封装*********************************
//  将二叉排序树调整为平衡二叉树
void AVLTreeFix(AVLTreeP *TreeRoot, int data, int type) {
    if (type == 1) { // 插入后进行调整
        if (updateBF(&(*TreeRoot)) >= 2) {  // 如果左子树高度比右子树高度大 2
            if (data < (*TreeRoot)->lchild->data) { // 如果是插入在左孩子的左子树，做 LL 调整
                LL_Rotation(&(*TreeRoot));
            } else {                                // 如果是插入在左孩子的右子树，做 LR 调整
                LR_Rotation(&(*TreeRoot));
            }
        }
```

```
        if (updateBF(&(*TreeRoot)) <= -2) { // 如果右子树高度比左子树高度大 2
            if (data > (*TreeRoot)->rchild->data) { // 如果是插入在右孩子的右子树，做 RR 调整
                RR_Rotation(&(*TreeRoot));
            } else {                                // 如果是插入在右孩子的左子树，做 RL 调整
                RL_Rotation(&(*TreeRoot));
            }
        }
    }
    if (type == 0) { // 删除后进行调整
        if (updateBF(&(*TreeRoot)) >= 2) {  // 如果左子树高度比右子树高度大 2
            // 删除与插入要判断的符号逆向
            if (updateBF(&((*TreeRoot)->lchild)) <= -1) { // 如果是删除左孩子的左子树，做 LR 调整
                LR_Rotation(&(*TreeRoot));
            } else {                                // 如果是删除左孩子的右子树，做 LL 调整
                LL_Rotation(&(*TreeRoot));
            }
        }
        if (updateBF(&(*TreeRoot)) <= -2) { // 如果右子树高度比左子树高度大 2
            if (updateBF(&((*TreeRoot)->rchild)) >= 1) { // 如果是删除右孩子的右子树，做 RL 调整
                RL_Rotation(&(*TreeRoot));
            } else {                                // 如果是删除右孩子的左子树，做 RR 调整
                RR_Rotation(&(*TreeRoot));
            }
        }
    }
}

//  返回右子树的最小值
int getRchildMin(AVLTreeP *TreeRoot) {
    if ((*TreeRoot)->lchild == NULL) {
        return (*TreeRoot)->data;
    } else {
        return getRchildMin(&((*TreeRoot)->lchild));
    }
}

// 计算节点的平衡因子
int updateBF(AVLTreeP *TreeRoot) {
    return GetHeight((*TreeRoot)->lchild) - GetHeight((*TreeRoot)->rchild);
}

// 求树的深度
int GetHeight(AVLTreeP TreeRoot) {
    if (TreeRoot == NULL) return 0;
    return 1 + max(GetHeight(TreeRoot->lchild), GetHeight(TreeRoot->rchild));
}

********************************各种调整*********************************
// 左旋转
void LL_Rotation(AVLTreeP *TreeRoot) {
    AVLTreeP temp = (*TreeRoot)->lchild;
    (*TreeRoot)->lchild = temp->rchild;
    temp->rchild = *TreeRoot;
    *TreeRoot = temp;
    (*TreeRoot)->bf = max(GetHeight((*TreeRoot)->lchild), GetHeight((*TreeRoot)->rchild));
    temp->bf = max(GetHeight(temp->lchild), GetHeight(temp->rchild));
}

// 先左旋，后右旋
void LR_Rotation(AVLTreeP *TreeRoot) {
    RR_Rotation(&((*TreeRoot)->lchild));
    LL_Rotation(&(*TreeRoot));
}

// 右旋转
void RR_Rotation(AVLTreeP *TreeRoot) {
    AVLTreeP temp = (*TreeRoot)->rchild;
    (*TreeRoot)->rchild = temp->lchild;
    temp->lchild = *TreeRoot;
    *TreeRoot = temp;
    (*TreeRoot)->bf = max(GetHeight((*TreeRoot)->lchild), GetHeight((*TreeRoot)->rchild));
    temp->bf = max(GetHeight(temp->lchild), GetHeight(temp->rchild));
}

// 先右旋，后左旋
void RL_Rotation(AVLTreeP *TreeRoot) {
    LL_Rotation(&((*TreeRoot)->rchild));
    RR_Rotation(&(*TreeRoot));
}
```

七、实验数据及结果分析

在测试中，构造了 100 个数据元素序列。

257 4 749 661 639 720 884 673 512 225

303 278 272 288 536 741 841 723 553 37

675 685 538 588 678 576 191 201 780 927

473 552 818 952 364 469 240 738 601 36

583 460 605 544 946 983 386 699 697 444

492 182 418 360 190 593 640 223 614 109

665 655 535 864 722 351 422 497 382 999

215 996 855 505 776 840 700 9 214 863

587 250 367 622 105 705 320 470 2 269

得到的排序二叉树如图 1 所示。

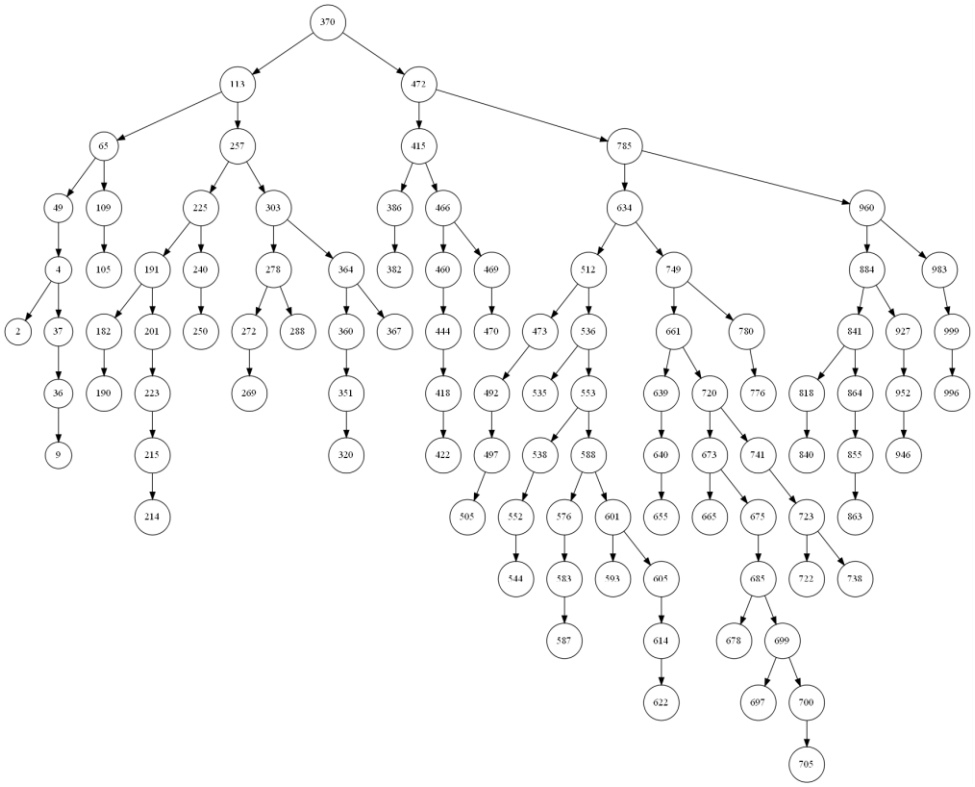


图 1 二叉排序树生成结果图

对于二叉排序树的删除，如果删除节点为叶子节点或者只含有一个子树，情况比较简单，这里就不截图展示，下面删除含有两个子树的节点，选择对根节点即 data 为 370 的节点进行删除。为了维持二叉排序树的排序性，算法会将待删除节点的右子树的最小值替代该节点的值，并且将待删除节点的右子树最小值所在的节点进行删除。删除前如图 2 所示。

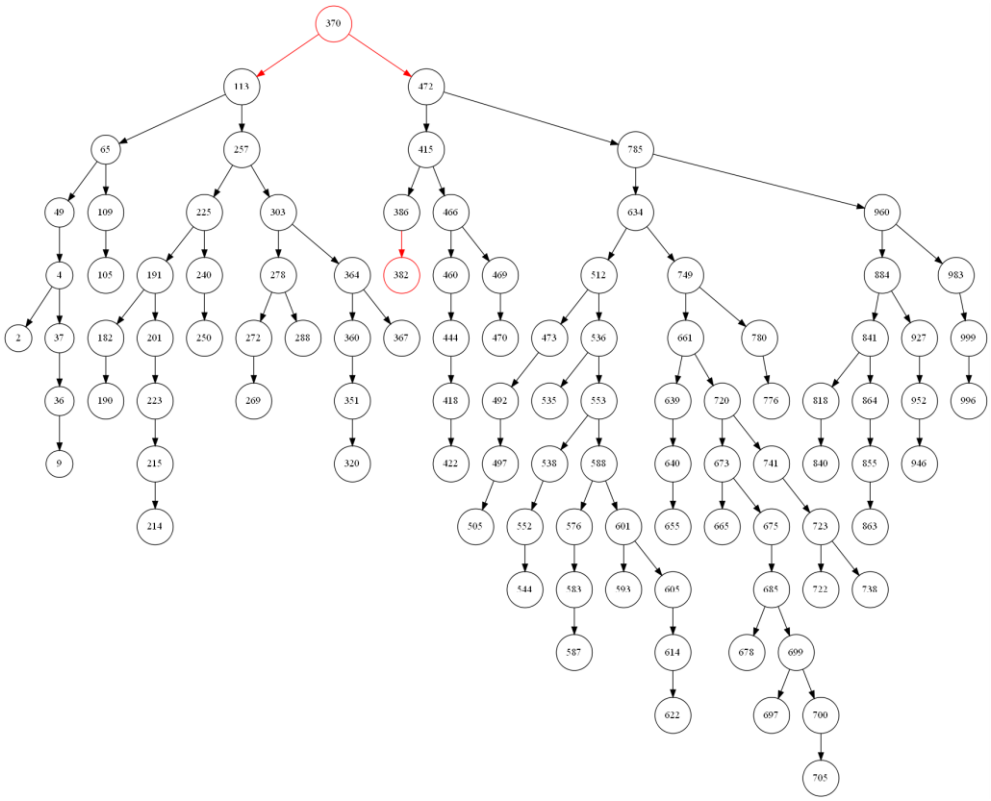


图 2　二叉排序树删除示意图

删除后的二叉排序树如图三所示。

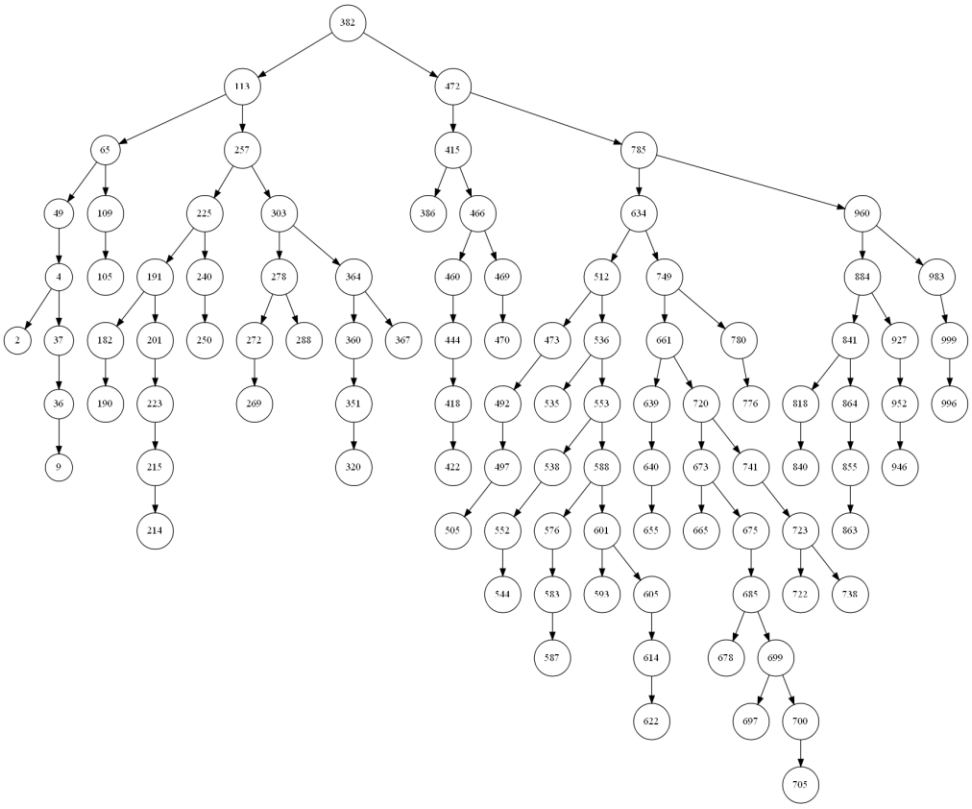


图 3　二叉排序树删除结果图

对于同样的输入序列，生成得到的平衡二叉树如图 4 所示。

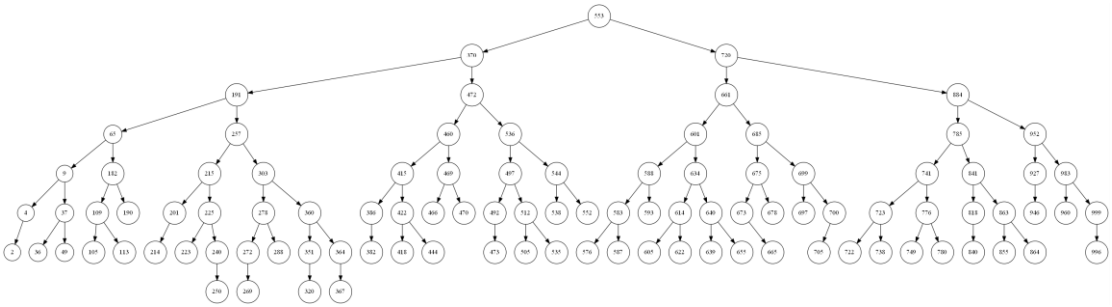


图 4　平衡二叉树生成结果图

对比图 1 和图 2，可以发现平衡二叉树的深度明显的比普通排序二叉树要小一些。平衡二叉树每一个节点的子树都是基本平衡的(最大相差 1)，这是平衡二叉树的定义所决定。普通排序二叉树的形状和输入数据非常相关，生成后的树的深度和平衡性不能得到保证。

对于平衡二叉树的删除，情况比较复杂。下面先从复杂的情况开始考虑。途中红色标注待删除的节点。

首先，在删除 data 为 269、288、367 的基础上删除 data 为 272 的节点，可以看成向 data 为 303 的节点的右子树的左子树插入节点导致的不平衡，做 RL 调整。

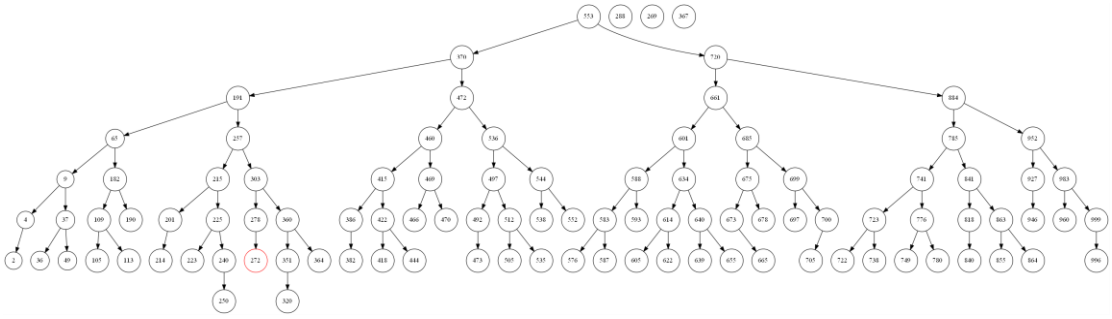


图 5　平衡二叉树 RL 删除前示意图

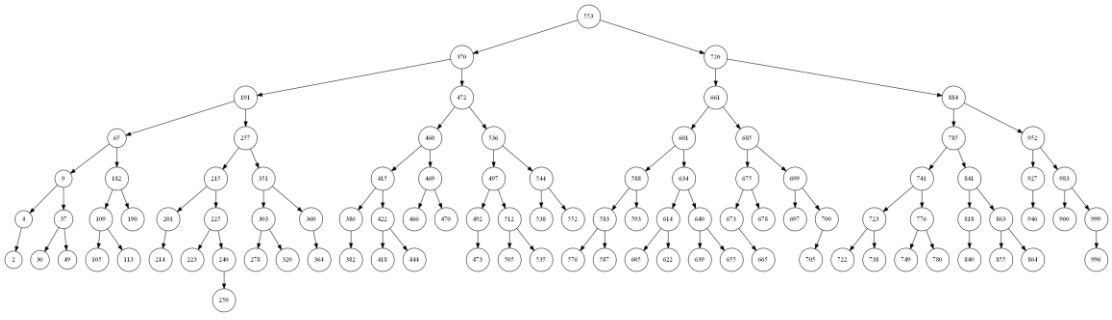


图 6　平衡二叉树 RL 删除后调整结果图

接着，在删除 data 为 250、418、466、382 的基础上删除 data 为 470 的节点，

可以看成向 data 为 460 的节点的左子树的右子树插入节点导致的不平衡，做 LR 调整。

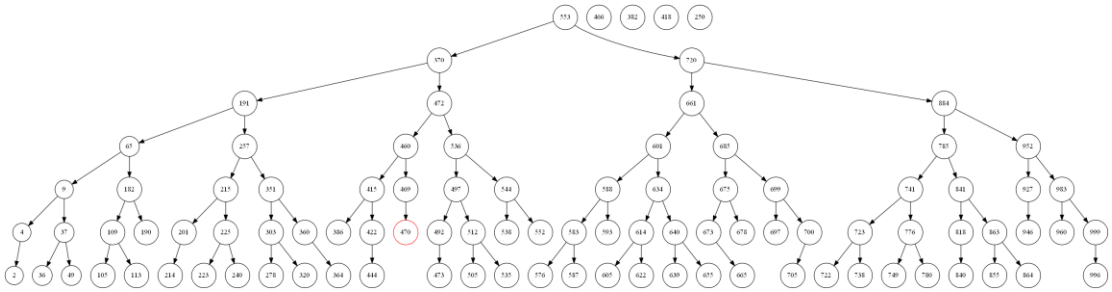


图 7　平衡二叉树 LR 删除前示意图

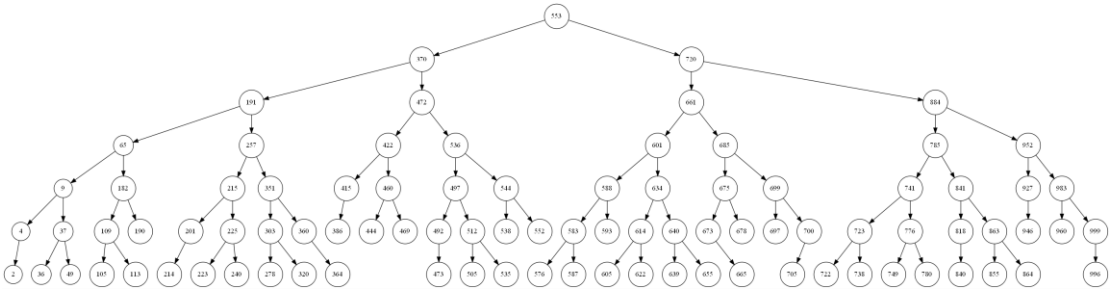


图 8　平衡二叉树 LR 删除后结果图

删除 data 为 697 的节点，其父节点为 data 为 699 的节点，可以看成向 data 为 699 的节点的右子树的右子树插入节点.导致的不平衡，做 RR 调整。

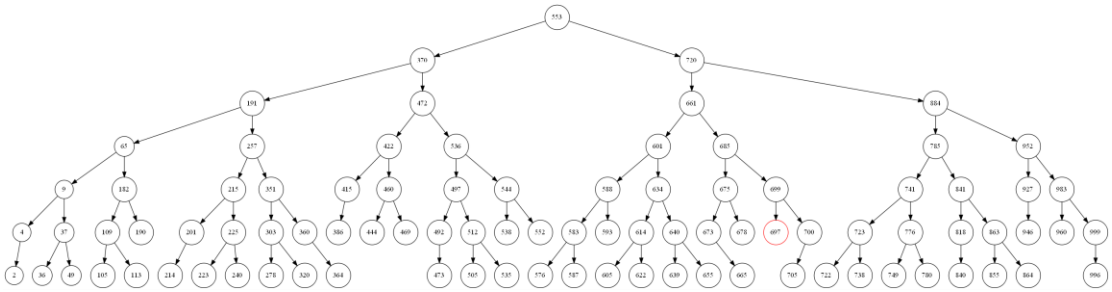


图 9　平衡二叉树 RR 删除前示意图

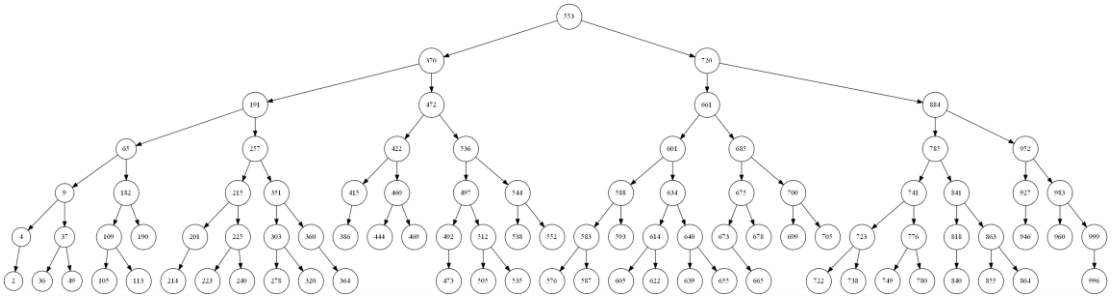


图 10　平衡二叉树 RR 删除后结果图

删除 data 为 678 的节点，其父节点为 data 为 675 的节点，可以看成向 data

的为 675 的节点的左孩子的左子树插入节点导致的不平衡，做 LL 调整。

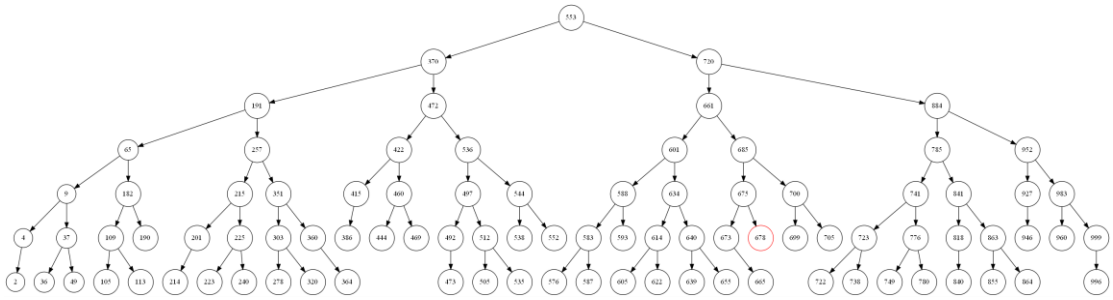


图 11　平衡二叉树 LL 删除前示意图

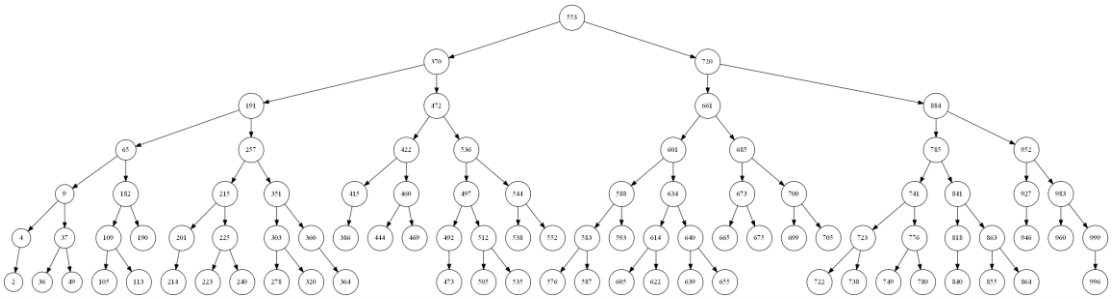


图 12　平衡二叉树 LL 删除后结果图

八、总结及心得体会：

本次试验成功实现了二叉排序树和平衡二叉树的增删查、遍历等相关功能。在实现这些功能的时候曾经想要实现的二叉排序树的修改，但是没有注意到二叉排序树是不能直接修改的节点的值，否则会破坏二叉排序树的排序性，对于修改排序树某个节点的值，我总结认为比较正确的方法应该是删除原有节点的值并插入新值，这是我没有考虑不周全导致的，也是其中一个缺点。

还有一个缺点是在实现二叉排序树的删除的时候，之前的思路是先调用二叉排序树的查找函数找到该节点的位置，再进行删除，逻辑比较混乱，导致后面写平衡二叉树删除的之后再维护平衡性时改写难度巨大，后来在实现的平衡二叉树删除的时候只好进行重构，递归查找并删除，提高了代码的健壮性和可维护性。

九、对本实验过程及方法、手段的改进建议：

我觉得 dot 语言实现可视化的时候给的文档不够全，类似像示例截图那样生成重复的 NULL 节点不知道如何实现，导致生成的图片可读性较差，比如左孩子生成到右孩子的位置，造成了误导，希望能够提供一份更好的文档，线上官方的文档有点繁琐。

电 子 科 技 大 学

实 验 报 告

实验二

一、实验室名称：

电子科技大学清水河校区主楼 A2-412

二、实验项目名称：

堆的应用：统计海量数据中最大的 K 个数（Top-K 问题）

三、实验内容和目的

实验内容：实现堆调整过程，构建小顶堆缓冲区，将海量数据读入依次和堆顶元素比较，若新元素小则丢弃，否则与堆顶元素互换并梳理堆保持为小顶堆。

实验目的：假设海量数据有 N 个记录，每个记录是一个 64 位非负整数。要求在最小的时间复杂度和最小的空间复杂度下完成找寻最大的 K 个记录。一般情况下，N 的单位是 G，K 的单位是 1K 以内，K<<N。

四、实验原理

没有必要对所有的 N 个记录都进行排序，我们只需要维护一个 K 个大小的数组，初始化放入 K 个记录。按照每个记录的统计次数由大到小排序，然后遍历这 N 条记录，每读一条记录就和数组最后一个值对比，如果小于这个值，那么继续遍历，否则，将数组中最后一条数据淘汰，加入当前的数组。最后当所有的数据都遍历完毕之后，那么这个数组中的 K 个值便是我们要找的 Top-K 了。不难分析出，这样，算法的最坏时间复杂度是 O(NK)。

在上述算法中，每次比较完成之后，需要的操作复杂度都是 K，因为要把元素插入到一个线性表之中，而且采用的是顺序比较。这里我们注意一下，该数组是有序的，一次我们每次查找的时候可以采用二分的方法查找，这样操作的复杂度就降到了 logK，可是，随之而来的问题就是数据移动，因为移动数据次数增多了。

利用堆进行优化。借助堆结构，我们可以在 log 量级的时间内查找和调整/

移动。因此到这里，我们的算法可以改进为这样，维护一个 K 大小的小根堆，然后遍历 N 个数据记录，分别和根元素进行对比。采用最小堆这种数据结构代替数组，把查找目标元素的时间复杂度有 O(K)降到了 O(logK)。最终的时间复杂度就降到了 O(NlogK)，性能改进明显改进。

实施过程：

(1) 构造小顶堆

由于堆是一棵完全二叉树，可以使用数组按照树的层次遍历的顺序存储堆。每次调整过程是末尾处开始，第一个含有子树的节点开始进行筛选调整，从下向上，每行从右往左。结束时，堆顶即为最值。

(2) 统计

1. 取出一个数据，与小顶堆的堆顶比较，若堆顶较小，则将其替换，重新调整一次堆；否则堆保持不变。

2. 继续再读一个缓冲区，重复上述过程。

3. 最终，堆中数据即为海量数据中最大的 K 个数。

堆调整的实现

1. 找到相应插入位置，同时记录离插入位置最近的可能失衡节点 A(A 的平衡因子不等于 0)。

2. 插入新节点 S。

五、实验器材（设备、元器件）

处理器：Intel® Core™ i7-6700HQ CPU @ 2.60GHz　2.60GHz

已安装的内存(RAM)：8GB

系统类型：64 位操作系统，基于 x64 的处理器

IDE：CLion 2018.2.4

Environment：mingw32 (Version: w64 6.0)

cMake：Bundled (Version: 3.12.3)

Debugger: MinGW-w64 GDB (Version: 8.1)

六、实验步骤

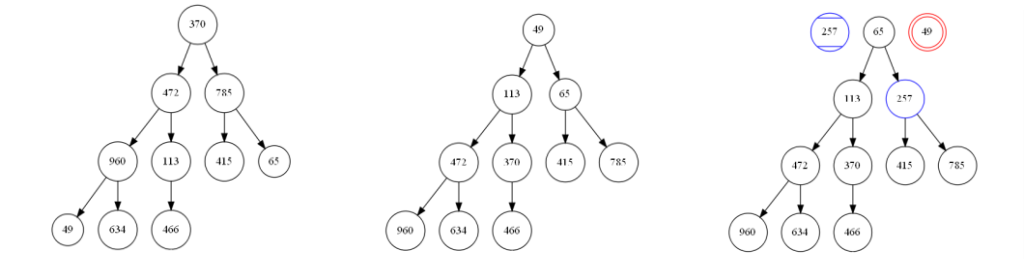
Top-K 算法

```c
**************************Top-K 问题*********************************
// 维护最大 k 个数的数组
void Top_k(int Tarray[], int Tsize, const int array[], int size) {
    // 构建初始小顶堆
    for (int i = 0; i < Tsize; i++) {
        Tarray[i] = array[i];
    }
    for (int i = Tsize / 2 - 1; i >= 0; i--) {
        HeapShift(Tarray, Tsize - 1, i);
    }
    // 维护小顶堆
    for (int j = Tsize; j < size; j++) {
        if (array[j] > Tarray[0]) {
            Tarray[0] = array[j];
            for (int i = Tsize / 2 - 1; i >= 0; i--) {
                HeapShift(Tarray, Tsize - 1, i);
            }
        }
    }
}

// 堆调整
void HeapShift(int array[], int size, int start) {
    int dad = start; // 父亲节点
    int son = 2 * dad + 1; // 儿子节点
    while (son <= size) {
        // 找子节点的最大值
        if (son + 1 <= size && array[son] > array[son + 1]) {
            son++;
        }
        if (array[dad] < array[son]) {
            return;
        } else {
            swap(&array[dad], &array[son]);
            // 查看交换是否导致子根结构混乱，若是，则向下调整；
            dad = son;
            son = 2 * dad + 1;
        }
    }
}
// 堆排序，从 0 开始编号
void HeapSort(int array[], int size) {
    // 构建初始堆
    for (int i = size / 2 - 1; i >= 0; i--) {
        HeapShift(array, size - 1, i);
    }
    for (int j = size - 1; j > 0; j--) {
        // 堆顶元素和堆中的最后一个元素交换
        swap(&array[0], &array[j]);
        // 重新调整结构，使其继续满足堆定义
        HeapShift(array, j - 1, 0);
    }
}

// // 找到最大的第 K 个数，leedcode - 215. Kth Largest Element in an Array
int findKthLargest(int *nums, int numsSize, int k) {
    int Tarray[k];
    for (int i = 0; i < k; i++) {
        Tarray[i] = nums[i];
    }
    for (int i = k / 2 - 1; i >= 0; i--) {
        HeapShift(Tarray, k - 1, i);
    }
    // 维护小顶堆
    for (int j = k; j < numsSize; j++) {
        if (nums[j] > Tarray[0]) {
            Tarray[0] = nums[j];
            for (int i = k / 2 - 1; i >= 0; i--) {
                HeapShift(Tarray, k - 1, i);
            }
        }
    }
    HeapSort(Tarray, k);
    Traversing(Tarray, k);
    return Tarray[k - 1];
}

// 交换
void swap(int *a, int *b) {
    *a = *a + *b;
```

```
    *b = *a - *b;
    *a = *a - *b;
}

// 遍历
void Traversing(int array[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", array[i]);
    }
    printf("\n");
}

// 可视化部分
void createDotFile(const char *filename, char *title, int *array, int size) {
    FILE *fp = fopen(filename, "w"); // 文件指针
    if (fp == NULL) {   // 为NULL则返回
        printf("File cannot open!");
        exit(0);
    }
    fprintf(fp, "digraph G {\n");   // 开头
    fprintf(fp, "graph[label=\"%s\"];", title); // 标题
    for (int i = 0; i < size; ++i) {
        fprintf(fp, "%d [shape=circle];\n", array[i]);
        int left_index = 2 * i + 1;   // 以索引0开头，左孩子的索引
        if (left_index < size) {
            fprintf(fp, "%d->%d;\n", array[i], array[left_index]);
        }
        int right_index = 2 * i + 2;  //以索引0开头，右孩子的索引

        if (right_index < size) {
            fprintf(fp, "%d->%d;\n", array[i], array[right_index]);
        }
    }
    fprintf(fp, "%d->struct[color=white];", array[size - 1]); // 看不见的线

    fprintf(fp, "struct [shape=record,label=\"{value|address}"); // 打印表格
    for (int k = 0; k < size; ++k) {
        fprintf(fp, "|{%d|%d}", array[k], k);
    }
    fprintf(fp, "\"];\n");
    fprintf(fp, "}\n"); // 结尾
    fclose(fp); // 关闭IO
}
```

七、实验数据及结果分析

在测试中，构造了堆调整的每个过程，如图 3 所示。



value	370	472	785	960	113	415	65	49	634	466
address	0	1	2	3	4	5	6	7	8	9

Init Heap

value	49	113	65	472	370	415	785	960	634	466
address	0	1	2	3	4	5	6	7	8	9

Adjust Heap

value	65	113	257	472	370	415	785	960	634	466
address	0	1	2	3	4	5	6	7	8	9

Time = 11

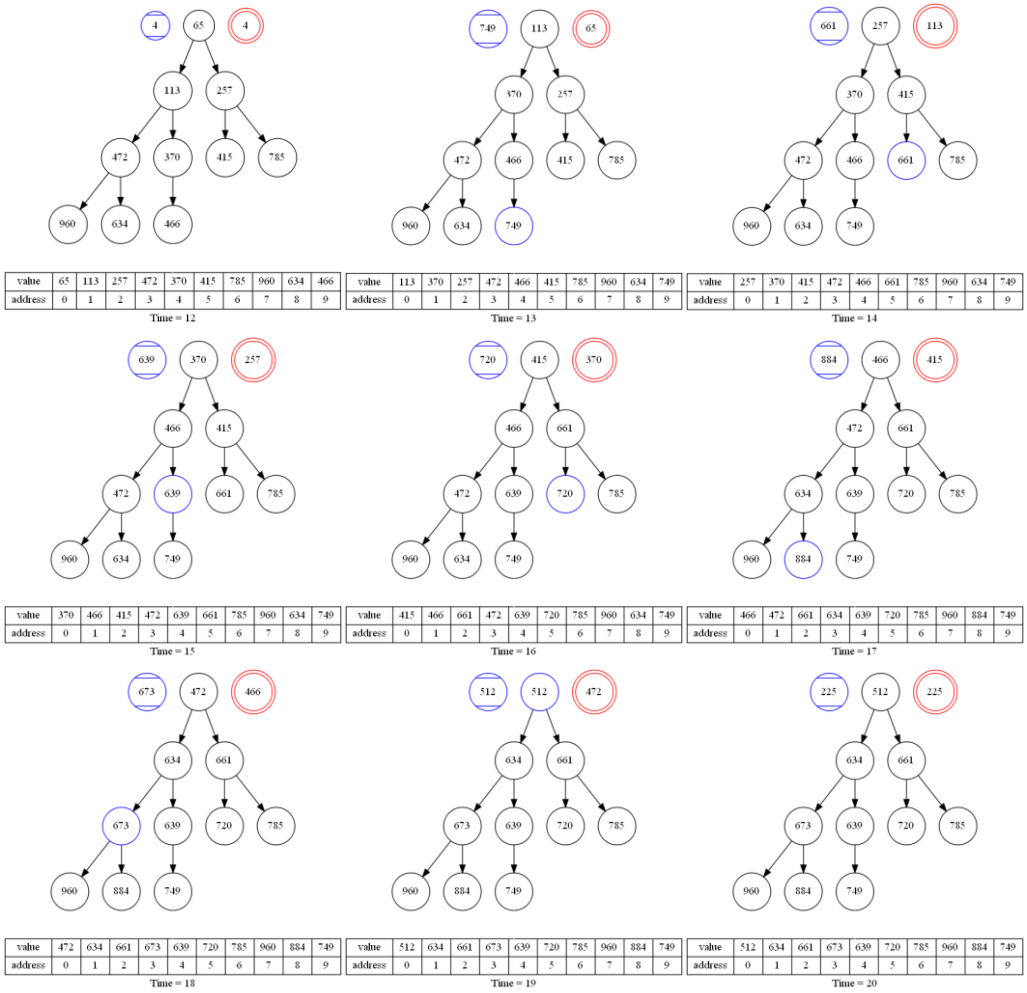


图 13　一次 Top10 的小顶堆调整过程

八、总结及心得体会：

本次实验实现了小顶堆的维护来解决 Top-K 问题，还写了一下堆排序。如果按照自己的纯理解开始写很有可能会用二叉树的逻辑结构来做这个实验，没有想到完全二叉树是可以用数组来表示的，使用二叉树的逻辑结构不够优美。另外在写小顶堆的时候参考了书上的代码，但是没有注意到书上的参考代码是从数组下标 1 开始构建小顶堆的，而我是用 0 开始，不同的下标计算的对于子节点的就计算方法不同，这个是一个需要注意的点。另外最初开始写堆排序的没有深刻理解堆排序的思想，它可能不是严格的排序算法，但是却的的确确实现了排序的功能，真正理解后感觉还是很奇妙的。

九、对本实验过程及方法、手段的改进建议：

Dot 好难用，花在可视化的时间都快赶上写代码的时间了。

电 子 科 技 大 学

实 验 报 告

实验三

二、实验室名称：

电子科技大学清水河校区主楼 A2-412

二、实验项目名称：

图的应用：单源最短路径 Dijkestra 算法

三、实验内容和目的

实验内容：有向图 G，给定输入的顶点数 *n* 和弧的数目 *e*，采用随机数生成器构造邻接矩阵。设计并实现单源最短路径 Dijkestra 算法。测试以 v0 节点为原点，将以 v0 为根的最短路径树生成并显示出来。

实验目的：完成图的单源最短路径算法，可视化算法过程。测试并检查是否过程和结果都正确。

四、实验原理

给定一个带权有向图 G=(V,E) ，其中每条边的权是一个非负实数。另外，还给定 V 中的一个顶点，称为源。现在我们要计算从源到所有其他各顶点的最短路径长度。这里的长度是指路上各边权之和。这个问题通常称为单源最短路径问题。

Dijkstra 算法实际上是动态规划的一种解决方案。它是一种按各顶点与源点 v 间的路径长度的递增次序，生成到各顶点的最短路径的算法。既先求出长度最短的一条最短路径，再参照它求出长度次短的一条最短路径，依次类推，直到从源点 v 到其它各顶点的最短路径全部求出为止。

具体地，将图 G 中所有的顶点 V 分成两个顶点集合 S 和 T。以 v 为源点已经确定了最短路径的终点并入 S 集合中，S 初始时只含顶点 v，T 则是尚未确定到源点 v 最短路径的顶点集合。然后每次从 T 集合中选择 S 集合点中到 T 路径最短的那个点，并加入到集合 S 中，并把这个点从集合 T 删除。直到 T 集合为

空为止。

具体步骤

1、选一顶点 v 为源点，并视从源点 v 出发的所有边为到各顶点的最短路径（确定数据结构：因为求的是最短路径，所以①就要用一个记录从源点 v 到其它各顶点的路径长度数组 dist[],开始时，dist 是源点 v 到顶点 i 的直接边长度，即 dist 中记录的是邻接阵的第 v 行。②设一个用来记录从源点到其它顶点的路径数组 path[],path 中存放路径上第 i 个顶点的前驱顶点）。

2、在上述的最短路径 dist[]中选一条最短的，并将其终点（即<v,k>）k 加入到集合 s 中。

3、调整 T 中各顶点到源点 v 的最短路径。因为当顶点 k 加入到集合 s 中后，源点 v 到 T 中剩余的其它顶点 j 就又增加了经过顶点 k 到达 j 的路径,这条路径可能要比源点 v 到 j 原来的最短的还要短。调整方法是比较 dist[k]+g[k,j]与 dist[j]，取其中的较小者。

4、再选出一个到源点 v 路径长度最小的顶点 k,从 T 中删去后加入 S 中，再回去到第三步，如此重复，直到集合 S 中的包含图 G 的所有顶点。

五、实验器材（设备、元器件）

处理器：Intel® Core™ i7-6700HQ CPU @ 2.60GHz　2.60GHz

已安装的内存(RAM)：8GB

系统类型：64 位操作系统，基于 x64 的处理器

IDE：CLion 2018.2.4

Environment：mingw32 (Version: w64 6.0)

cMake：Bundled (Version: 3.12.3)

Debugger: MinGW-w64 GDB (Version: 8.1)

六、实验步骤

单源最短路径 Dijkestra 算法

```
1.  **********单源最短路径 Dijkestra 算法**********
2.  #define VEX_NUM 10  // 节点个数
3.  #define MAXINT 100000 // 定义无穷大
4.  typedef int dataType;
5.  typedef struct graph {
6.      char vexs[VEX_NUM]; // 顶点序列 VEX_NUM 为顶点数目
7.      dataType arcs[VEX_NUM][VEX_NUM];  // 邻接矩阵
8.  } Mgraph, *MgraphP;
9.  // 初始化邻接矩阵
10. void graphInit(MgraphP *graph, int row, int col) {
11.     int size = 100;
12.     DataType A[size];
13.     // 读取文件数据，保存在数组 A 中
```

```
    FILE *fp = fopen("./data/distance.txt", "r+");
    if (fp == NULL)
        exit(0x01);
    DataType keynumber;
    for (int i = 0; i < size; i++) {
        fscanf(fp, "%d", &keynumber);
        A[i] = keynumber;
    }
    fclose(fp);
    // 将读取的数据赋值到 graph 中
    for (int i = 0; i < row; i++) {
        for (int j = 0; j < col; j++) {
            (*graph)->arcs[i][j] = A[i * 10 + j];
        }
    }
}
// 初始化图
MgraphP graphCreate() {
    Mgraph *graph = (Mgraph *) malloc(sizeof(Mgraph)); // 申请节点空间
    if (!graph) { // 判断是否有足够的内存空间
        printf("申请内存空间失败！\n");
    }
    return graph;
}
// 单源最短路径
void Dijkstra(MgraphP graph, int v0, int *path, dataType *dist) {
    for (int i = 0; i < VEX_NUM; i++) {
        // 初始化距离
        dist[i] = graph->arcs[v0][i];
        // 如果不可达，则前驱置为-1
        if (dist[i] < MAXINT) { // 如果可达，则前驱置为该单源初始点
            path[i] = v0;
        } else {  // 如果不可达，则前驱置为-1
            path[i] = -1;
        }
    }
    int s[VEX_NUM] = {0}; // 记录没有遍历到的节点，初始为0
    dist[v0] = 0; // 节点到自身的路径长度为0
    s[v0] = 1; // 该节点已经遍历完成，置为1
    for (int i = 1; i < VEX_NUM; i++) { // 循环剩余 n-1 个节点
        int v = -1; // 记录点的位置，初始化为-1
        int min = MAXINT; // 记录点的最小值，初始化为最大值
        for (int j = 0; j < VEX_NUM; j++) {
            if (s[j] != 1 && dist[j] < min) { // 该节点未被遍历，并且该值最小
                min = dist[j]; // 记录该值
                v = j;  // 记录该点位置
            }
        }
        // 如果不存在该值，则跳过
        if (v == -1) {
            break;
        }
        // 否则进行下面
        s[v] = 1; // 表示该点已经遍历
        for (int j = 0; j < VEX_NUM; j++) {
            // 如果该点没被遍历，并且更新后的路径比原路径短，则替换
            if (s[j] != 1 && (min + graph->arcs[v][j] < dist[j])) {
                dist[j] = min + graph->arcs[v][j];
                path[j] = v;
            }
        }
    }
}
// 遍历邻接矩阵
void Traversing(dataType *array, int row, int col) {
    for (int i = 0; i < row; i++) {
        printf("v%d          ", i);
        for (int j = 0; j < col; j++) {
            printf("%d ", *(array + i * col + j));
        }
        printf("\n");
    }
}
// 打印该节点到各节点的最短路径
void printPath(int v0, int *path, int *dist) {
    for (int i = 0; i < VEX_NUM; i++) {
        // 不打印该节点到本节点的最短路径
        if (v0 != i) {
            printf("v%d -> v%d, minDist: %d, path: v%d <- ", v0, i, dist[i], i);
            int temp = path[i];
            // 如果该节点的前驱不等于源节点，则继续递归访问
            while (v0 != temp) {
                printf("v%d <- ", temp);
                // 记录前驱
                temp = path[temp];
            }
```

```
            printf("v%d", v0);
            printf("\n");
        }
    }
}
// 可视化部分
void createPathFile(MgraphP *graph, int v0, int *path, dataType *dist) {
    for (int i = 0; i < VEX_NUM; i++) {
        if (v0 != i) {
            // 创建文件
            char *filename = (char *) malloc(100);
            sprintf(filename, "./dotFile/v%d_to_v%d.dot", v0, i);
            FILE *fp = fopen(filename, "w"); // 文件指针
            if (fp == NULL) {   // 为NULL则返回
                printf("File cannot open!");
                exit(0);
            }
            char *title = (char *) malloc(100);
            sprintf(title, "v%d -> v%d", v0, i);    // 文件名
            fprintf(fp, "digraph G {\n");   // 开头
            // 构建原始领接图
            for (int j = 0; j < VEX_NUM; j++) {
                fprintf(fp, "v%d[shape=ellipse];\n", j);
                for (int k = 0; k < VEX_NUM; k++) {
                    if ((*graph)->arcs[j][k] < MAXINT) {
                        fprintf(fp, "v%d->v%d[label=\"%d\"];\n", j, k, (*graph)->arcs[j][k]);
                    }
                }
            }
            fprintf(fp, "graph[label=\"%s, minDist: %d\"];\n", title, dist[i]); // 标题生成，记录最短距离
            fprintf(fp, "edge[labelfontcolor=red,fontcolor=red,color=red];\n");
            int temp = path[i];
            // 初始路径
            fprintf(fp, "v%d[color=red];\n", i);
            fprintf(fp, "v%d->v%d[color=red,label=\"%d\"];\n", temp, i, (*graph)->arcs[temp][i]);
            // 遍历前驱路径并标红
            while (v0 != temp) {
                fprintf(fp, "v%d[color=red];\n", temp);
                fprintf(fp, "v%d->v%d[label=\"%d\"];\n", path[temp], temp,
                        (*graph)->arcs[path[temp]][temp]);
                temp = path[temp];
            }
            // 最终节点标红
            fprintf(fp, "v%d[color=red];\n", v0);
            fprintf(fp, "}\n"); // 结尾
            fclose(fp); // 关闭IO
            char *order = (char *) malloc(100);
            sprintf(order, "dot -Tpng ./dotFile/v%d_to_v%d.dot -o ./dotFile/v%d_to_v%d.png", v0, i, v0, i);
            system(order);
        }
    }
}
```

七、实验数据及结果分析

表 1　邻接矩阵

	v0	v1	v2	v3	v4	v5
v0	100000	1	8	9	100000	7
v1	7	100000	100000	12	9	1
v2	9	15	100000	9	9	100000
v3	14	16	1	100000	14	14
v4	100000	1	100000	100000	100000	100000
v5	100000	14	11	4	11	100000

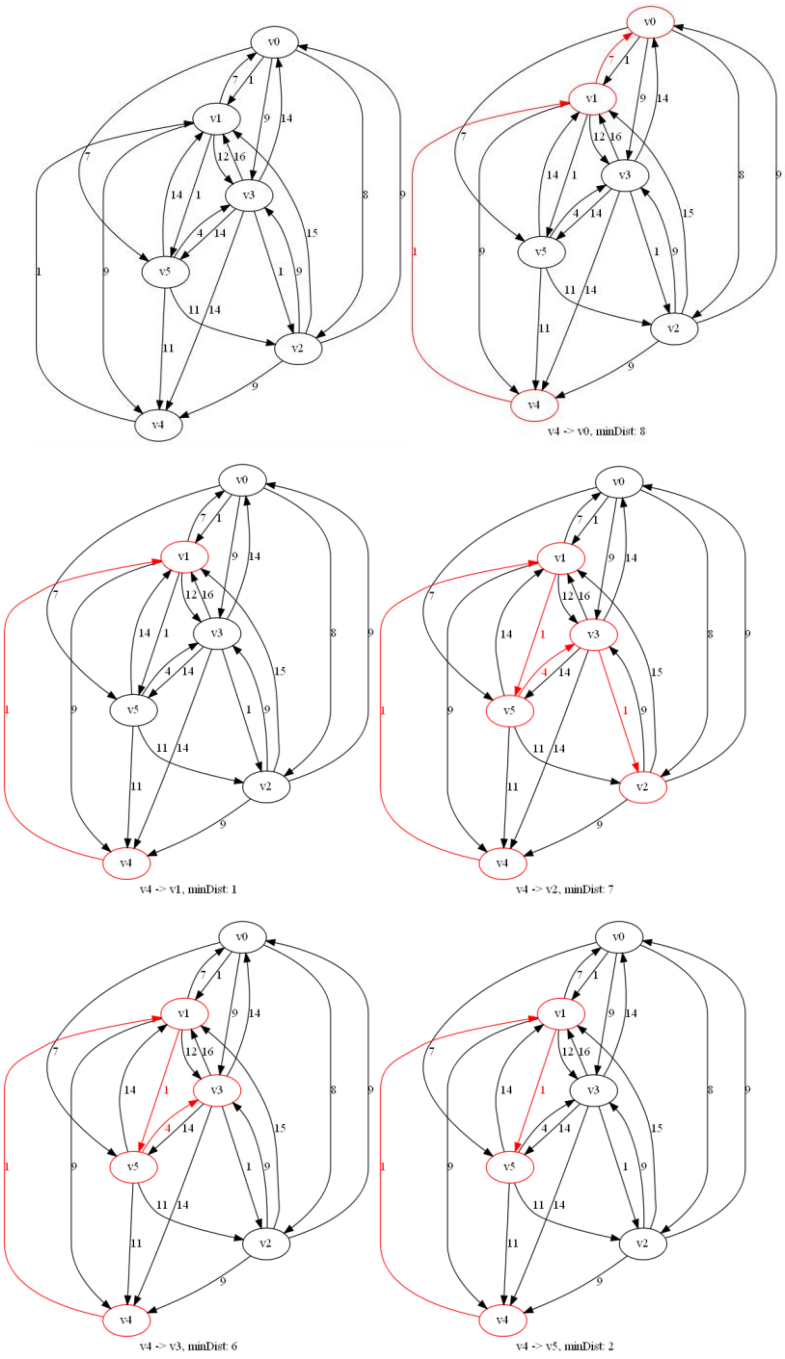


图 14　一次 6 个节点有向图最短路径算法过程

八、总结及心得体会：

本次实验使用了随机数来生成领接矩阵，导致有向图的形状会有些难看，可读性较差，但仍然可以体现算法功能的一般性，若要提高可读性，可以预先设置好拓扑图的形状，或者生成对称矩阵的无向图，这样拓扑图看起来会没那么复杂。在写代码的时候没有考虑后面可视化的问题，导致封装性较差，有待改进。自认为 dot 文件写的有点烂，没搞懂为什么同样的节点后面的样式会覆写前面的样式，而对于弧而言却是前后写的同样的弧会生成两条。

九、对本实验过程及方法、手段的改进建议：

希望下次能把用作示例的实验截图的实现写在文档里面，这样可以使学生更加专注于算法实验本身而不是在可视化。