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

学生姓名：李昊霖

学　　号：**2019270102007**

指导教师：周益民

日　　期：**2021** 年 **11** 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) 二叉树节点插入

在二叉排序树中插入新结点，要保证插入后的二叉树仍符合二叉排序树的定义。由于二叉搜索树的特殊性质确定了二叉排序树中每个元素只可能出现一次，所以在插入的过程中首先进行查找，如果发现这个元素已经存在于二叉搜索树中，就不进行插入。否则就查找合适的位置进行插入。

插入过程：若二叉排序树为空，则待插入结点*S 作为根结点插入到空树中；当非空时，将待插结点关键字 S->key 和树根关键字 t->key 进行比较，若 s->key = t->key,则无须插入，若 s->key< t->key,则插入到根的左子树中，若 s->key> t->key,则插入到根的右子树中。而子树中的插入过程和在树中的插入过程相同，如此进行下去，直到把结点*s 作为一个新的树叶插入到二叉排序树中，或者直到发现树已有相同关键字的结点为止。

(2) 生成了一棵二叉排序树

生成二叉排序树的过程就是一个不断查找并添加叶子节点的过程，有如下特点：

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。

平衡二叉树的实现

平衡二叉树的性质决定了其左右子树深度差至多为 1，因此插入节点后可能会导致不平衡，即左右子树深度差超过 1，此时需要对其进行平衡化操作。插入过程如下：

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。

3. 确定节点 B(B 是失衡节点 A 的其中一个孩子，就是在 B 这支插入节点导

致的不平衡，但是 B 的平衡因子为-1 或 1)

4. 修改从 B 到 S 路径上所有节点的平衡因子。(这些节点原值必须为 0，如果不是，A 值将下移)。

5. 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。

每插入一个结点就进行调整使之平衡。调整策略，根据二叉排序树失去平衡的不同原因共有四种调整方法。

第一种情况：LL 型平衡旋转

由于在 A 的左子树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。例子如下图 1-a：

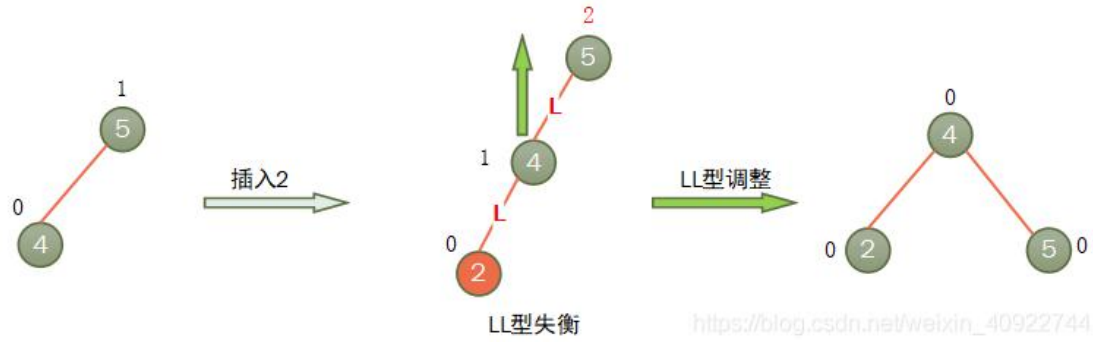


图 1-a　LL 型失衡

第二种情况：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。例子如下图 1-b：

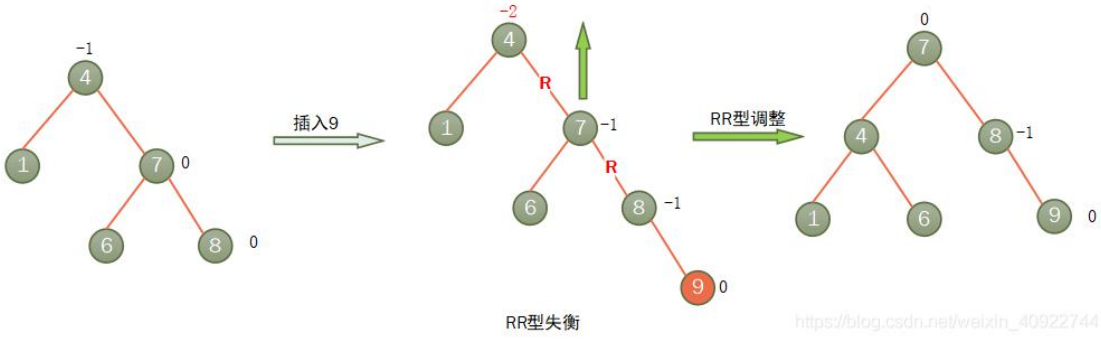


图 1-b　RR 型失衡

第三种情况：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。例子如下图 1-c：

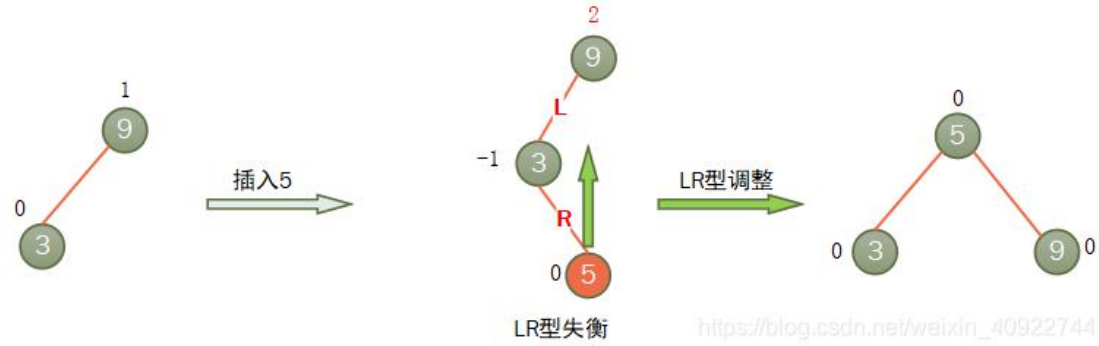


图 1-c　LR 型失衡

第四种情况：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。例子如下图 1-d：

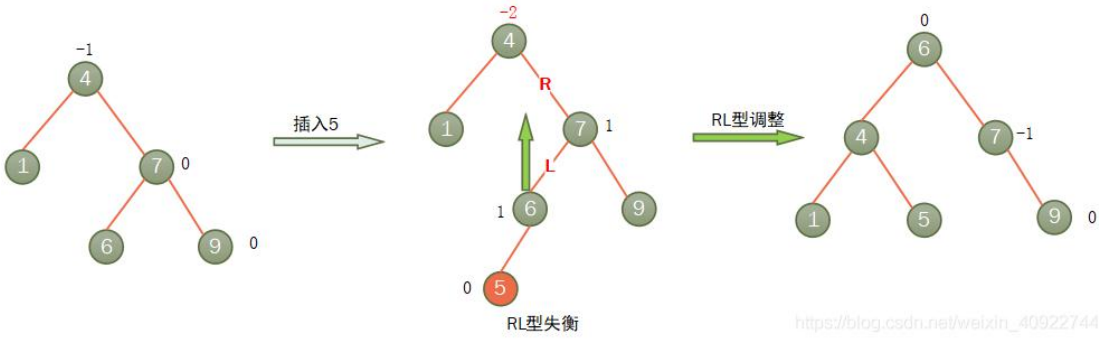


图 1-d　RL 型失衡

五、实验器材（设备、元器件）

硬件平台：处理器: Intel(R) Core(TM) i7-9750H CPU @2.60GHz (12 CPUs), ~2592MHz　内存:16384MB RAM　　外存：512GB SSD　　显　卡　：　NVIDIA GeForce GTX 1660 Ti

软件平台：操作系统: Windows 10 家庭中文版 64 位(10.0，内部版本 19043)

IDE：Dev-C++ 5.11.0.0　　编译器：gcc version 8.1.0

六、实验步骤

二叉排序树，二叉平衡树

```
#define MAXSIZE 50
#define LH +1
#define EH 0
#define RH -1
#define TRUE 1
#define FALSE 0
const int N=1000;
```

```
#define MAX 5 /* 字符串最大长度+1 */
typedef int KeyType; /* 设关键字域为整型*/
#define LEN sizeof(HFtree)  //HFtree 结构的大小
//二叉搜索树，二叉排序树
typedef struct node
{
    int bf;    //平衡因子
    int data;
    struct node *lchild,*rchild;

}BSTNode;
/*************************************************************
**/
void PrintBST_1(BSTNode *bt);//非递归中序遍历平衡二叉树
void PrintBST_2(BSTNode *bt);//递归打印二叉搜索树
/*************************************************************
**/
BSTNode *SearchBST(BSTNode *bt,int k,BSTNode *p,BSTNode *&f);//f 返回的
是父亲节点，p 是父节点
/*************************************************************
**/
void DeleteBSTNode(BSTNode *p,BSTNode *&r);////删除某个节点具体实现
void Delete(BSTNode *&bt);//删除节点对全树调整
int DeleteBST(BSTNode * &bt,int k);//删除平衡二叉树某节点实现
/*************************************************************
**/
void CreateArray(int *Array,int n);//建立数组
void InputArray(int * &Array,int n);//接收数组输入
void Swap(int *a,int *b);//交换节点
/*************************************************************
**/
void R_Rotate(BSTNode *&p);//右旋
void L_Rotate(BSTNode *&p);//左旋
void LeftBalance(BSTNode *&p);//LR 旋
void RightBalance(BSTNode *&p);//RL 旋
int InsertAVL(BSTNode *&T,int e,bool &taller);//插入 AVL 树节点
BSTNode *CreateAVL(int *A,int n);//生成 AVL 树
```

二叉排序树：根据测试数据，选择合适的查找数值，修改相应的函数 SearchBST 的参数并画图。

平衡二叉树：在 AVL.exe 中加入代码：system("dot.exe -Tpng AVL.gv -o AVL1.png"); 得到平衡二叉树的生成图。

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，以-1 作为结束标记。263 591 116 462 118 598 81 915 802 753 79 765 921 811 265 832 686 205 478 342 267 758 241 658 739 278 306 988 216 735 237 755 424 735 666 225 876 581 877 351 653 953 764 716 176 572 912 879 184 452 987 672 799 76 288 370 383

346 31 936 392 960 831 146 335 14 274 521 709 814 117 120 713 240 202 634 314 426 687 371 614 978 495 268 582 265 191 867 378 659 80 769 852 163 805 762 266 202 806 485 -1

得到的排序二叉树如图二所示。

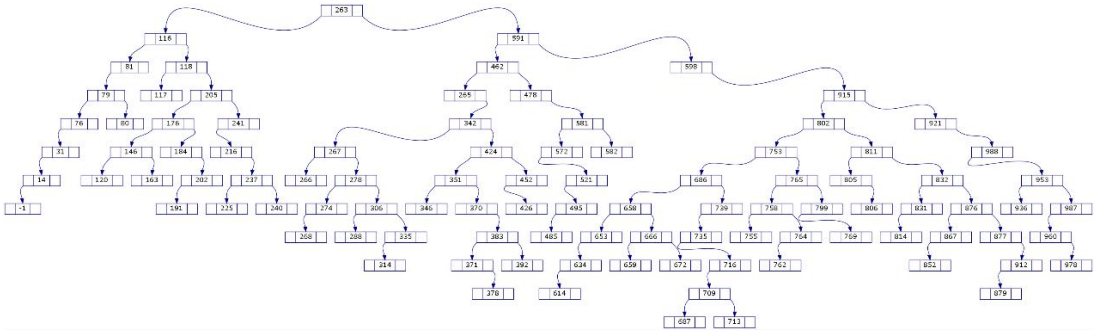


图 2 排序二叉树生成结果图

针对该组数据，在排序二叉树中查找了最深的节点“713”与一个不存在的数“999”，得到的查找结果如下图 3，4。

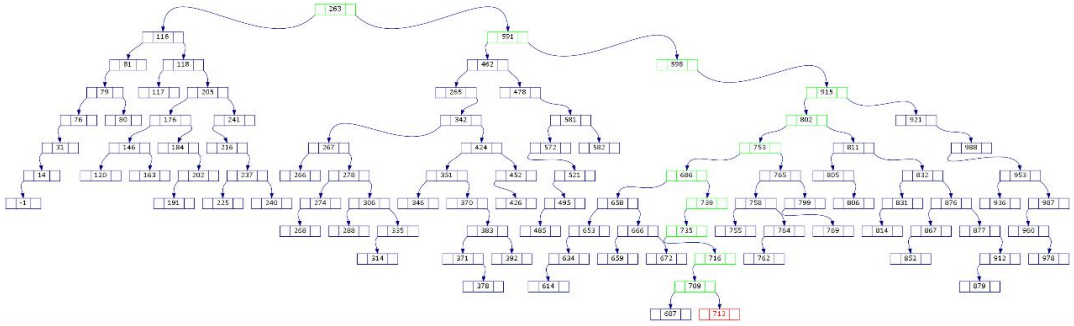


图 3 查找 713 结果图

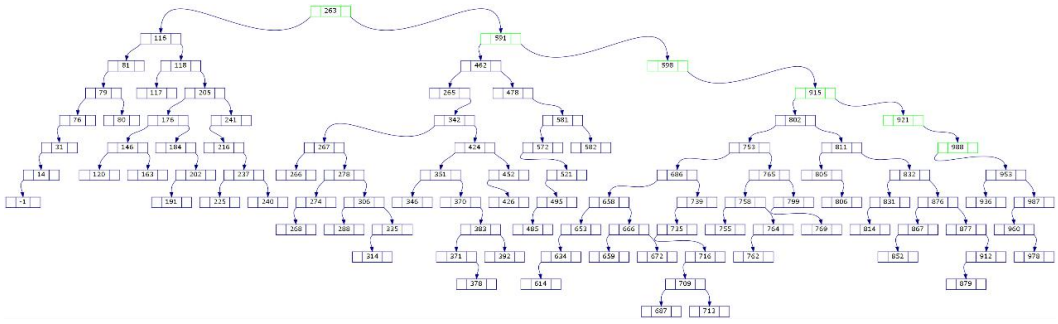


图 4 查找 999 结果图

对于同样的输入序列，生成得到的平衡二叉树如图 5 所示。

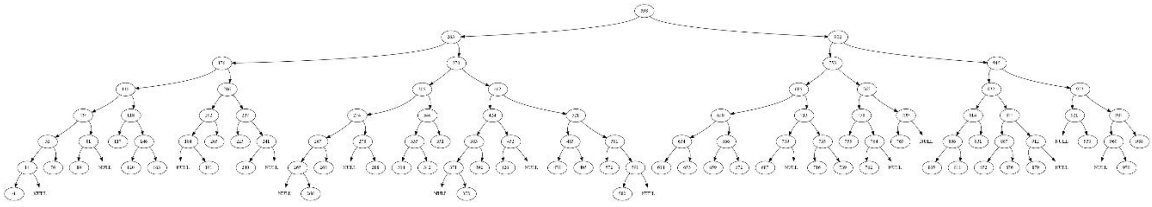


图 5 平衡二叉树生成结果图

再设计一组有序的数据（及其逆序）：1 2 3 4 5 6 7 8 9 10 11 12 14 17 19 33 66 77 88 99 102 345 786 生成排序二叉树，得到链表形式的二叉排序树，如下图六、七。

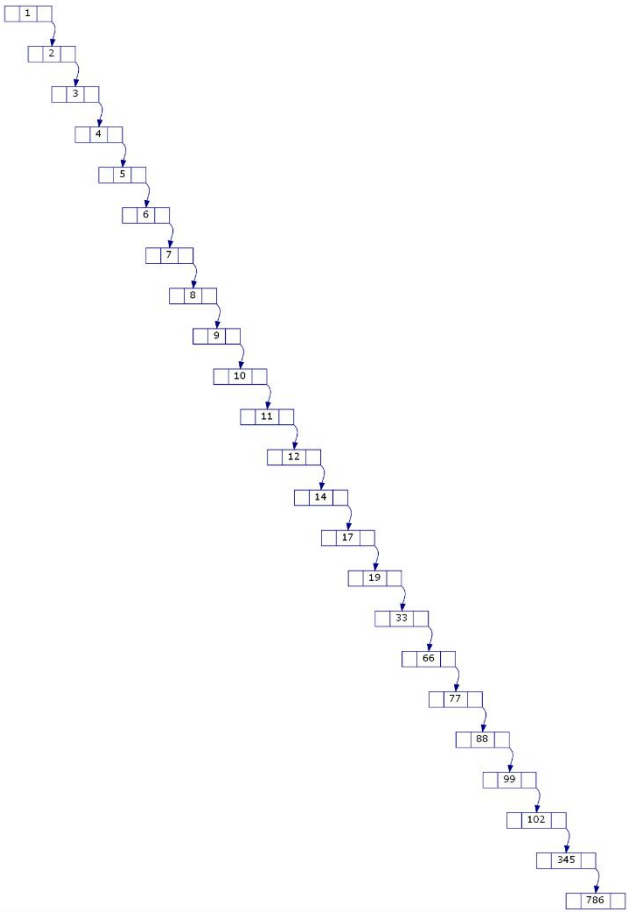


图 6 退化的二叉排序树

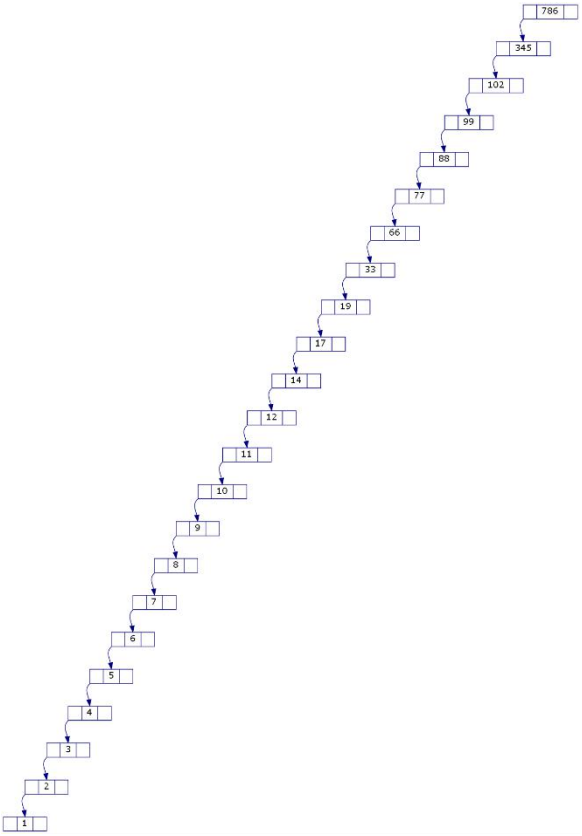


图 7 退化的二叉排序树

该组数据生成的平衡二叉树如下图 8.

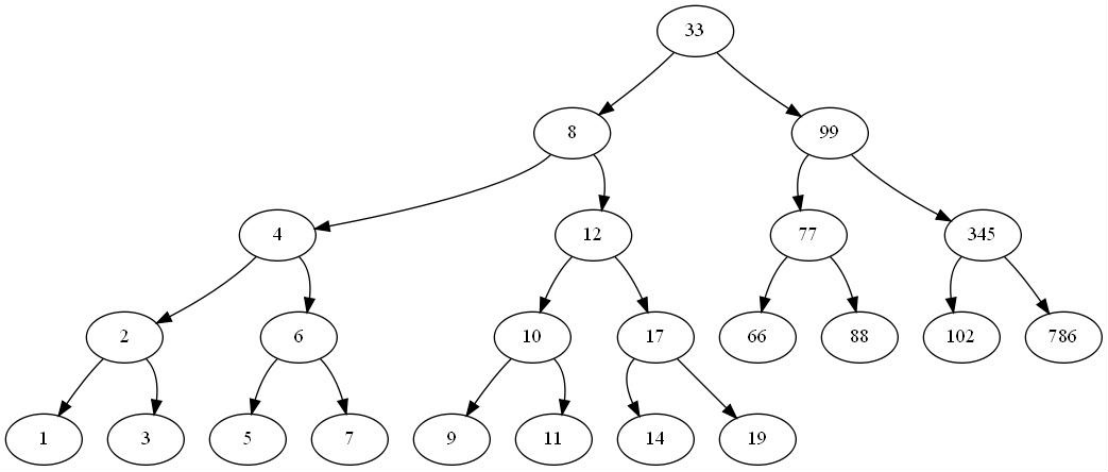


图 8 平衡二叉树生成图

对比图 1 和图 5，图 6 和图 8，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。

对比图 1 和图 6，图 7，可以看出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。图 6，7 说明了当输入数据有序时排序二叉树退化为链表。

分析图3和图4可以看出,二叉排序树的查找速度与所查找节点的深度相关，节点的深度越深，查找所消耗的时间越长。即使对于不存在的节点（例如图4查找999），也需要找到它的临近节点。查找算法的平均时间复杂度为O(log n)。

八、总结及心得体会：

二叉排序树来组织数据,由于二叉排序树是应用折半查寻法思想进行对数据进行存储的，所以，其左孩子大于双亲结点、右孩子小于双亲结点（或者左孩子小于双亲结点、右孩子大于双亲结点），这样就可以应用折半查寻法的思想进行查寻，从而减少对排序时所消耗的时间。

但二叉排序树对输入数据较为敏感，若输入数据有序，其退化为链表，查找的最坏时间复杂度为O（n）；若输入数据随机，则其性能较好。

平衡二叉树是对二叉搜索树的一种改进。二叉搜索树有一个缺点就是，树的结构是无法预料的，随意性很大，它只与节点的值和插入的顺序有关系，往往得到的是一个不平衡的二叉树。在最坏的情况下，可能得到的是一个单支二叉树，其高度和节点数相同，相当于一个单链表，对其正常的时间复杂度有O(log n)变成了O(n)，从而丧失了二叉排序树的一些应该有的优点。平衡二叉树则没有这个问题，对输入数据的泛化性较好，即使最坏情况下也可以保证O(log n)的时间复杂度。

但是，相应地，平衡二叉树的维护更麻烦，在插入（删除）新节点时需要针对不同的失衡情况进行旋转，以保证树的平衡。它优异的查找性能是建立在更大的维护开销上的。

九、对本实验过程及方法、手段的改进建议：

1.给出详细的对输入数据的要求，例如：二叉排序树添加一组有序输入，验证特殊情况，与一般情况对比。

2.添加随机数生成程序，方便使用

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

最直接的方法自然是对数据进行排序，但这样做显然需要消耗大量的时间复杂度，对海量数据显然不合适。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大小为 K 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。

上述算法的最坏时间复杂度为 O(NK)，针对 N 显然无法再优化，因为无论如何都要遍历至少一次数组，而对于 K，可以通过改变所维护数组的结构进行优化。对于数据量为 K 的数组，每次只需比较当前扫描的数据与数组中最小的元素，若其大于数组中最小的元素，则更新；反之继续扫描。因此可以采用小顶堆的结构维护该数组。这样我们可以在 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 大小的小顶堆，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由 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) 构造堆

由于堆是一棵完全二叉树，可以使用数组按照广度优先的顺序存储堆。每次调整过程是末尾处开始，第一个非终端结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）

硬件平台：处理器: Intel(R) Core(TM) i7-9750H CPU @2.60GHz (12 CPUs), ~2592MHz 内存:16384MB RAM 外存：512GB SSD 显卡：NVIDIA GeForce GTX 1660 Ti

软件平台：操作系统: Windows 10 家庭中文版 64 位(10.0，内部版本 19043)

IDE：Dev-C++ 5.11.0.0 编译器：gcc version 8.1.0

六、实验步骤

实验源代码根据所提供的 Heap.cpp 修改。在实验过程中发现代码中的画图模块严重影响了运行时间，故在比较运行时间与 K，数据量 N 的关系时注释掉了代码中的画图部分，如下图 1。

```
sprintf(figlabel, "Input %d Drop %d",temp);
if(temp > Heap[0])
{
    int pretop = Heap[0];
    Heap[0] = temp;
    HeapAdjust(Heap,0,Nnum);
    //sprintf(figlabel, "Time = %d",i);
    //DotHeap(Heap,Nnum,figlabel,temp,pretop);
}
else
{
    //sprintf(figlabel, "Time = %d",i);
    //DotHeap(Heap,Nnum,figlabel,temp,temp);
}

//sprintf(orderstr, "dot.exe -Tpng heapT.gv -o Tree%02d.png",i++);
//system(orderstr);
}
```

图 1 注释后的代码

为了计算程序运行时间，添加了<windows.h>库，使用QueryPerformanceCounter函数计算时间，该函数可精确到1ms，在计算运行时间的函数中性能较好，具体代码如下图2:

```
double time=0;
double counts=0;
LARGE_INTEGER nFreq;
LARGE_INTEGER nBeginTime;
LARGE_INTEGER nEndTime;
QueryPerformanceFrequency(&nFreq);
QueryPerformanceCounter(&nBeginTime);//开始计时

sprintf(figlabel, "Current Heap Situation");
DotHeap(Heap,Nnum,figlabel,temp);

QueryPerformanceCounter(&nEndTime);//停止计时
time=(double)(nEndTime.QuadPart-nBeginTime.QuadPart)/(double)nFreq.QuadPart;//计算程序执行时间单位为s
cout<<"Heapsize(K): "<<Heapsize<<endl;
cout<<"运行时间: "<<time*1000<<"ms"<<endl;

sprintf(orderstr, "dot.exe -Tpng heapT.gv -o FinaT.png");
system(orderstr);
```

图 2 计算运行时间代码

实验还需要生成随机数，这里采用python的numpy库实现随机数生成。

完成上述对代码的修改后进行使用。两组对比实验分别固定数据总量N与K中的一个，改变另一个值的大小，探究两者与代码运行时间的关系，并绘图展示。在不断实验的过程中发现了一些问题，故添加一组实验探究随机数组成与运行时间的关系。最后，绘制一次K=12的小顶堆调整过程。

七、实验数据及结果分析

1.探究 N 与运行时间的关系：

固定 k=10，测试数据在[0，1000)范围内，以数据总量 N 作为横坐标（N 分别取值:），ln（t）为纵坐标，使用 matlab 绘制散点图 3 所示。

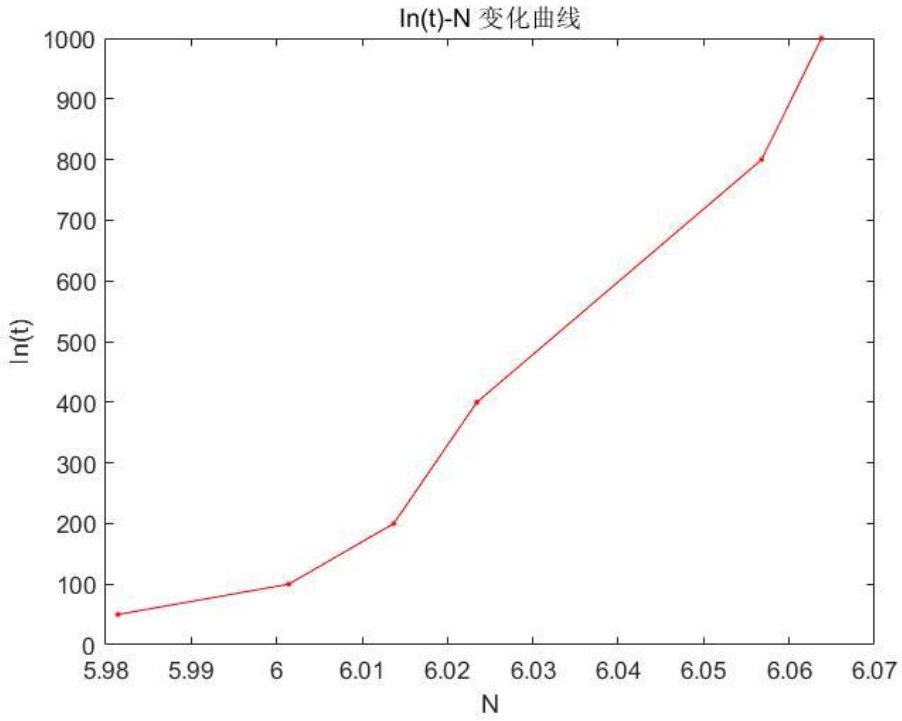


图 3　k=10 时，ln t 随 N 的变化曲线

如图所示，在 k 值确定时，时间 t 的对数与 N 基本呈线性关系，但数据有误差，可见耗时与数据量在误差允许的范围内成正比。

2.探究 K 与运行时间的关系：

实验数据为 100 个[0，100)范围内的测试数据（详细值见后），不断改变 K 的值进行实验，比较程序运行时间。这里选用了 K=5，6，7，8，9，10，15，20，25 时的数据。当 K<=3 时，堆最多只有两层，个人认为数据参考价值不大。

原始数据如下图 4：

```
D:\教材\数据结构\交付学生实验素材\Heap>Heap.exe num.txt
Heapsize(K)：4
运行时间：318.563ms
D:\教材\数据结构\交付学生实验素材\Heap>Heap.exe num.txt
Heapsize(K)：5
运行时间：333.445ms

D:\教材\数据结构\交付学生实验素材\Heap>Heap.exe num.txt
Heapsize(K)：6
运行时间：350.144ms
D:\教材\数据结构\交付学生实验素材\Heap>Heap.exe num.txt
Heapsize(K)：7
运行时间：359.783ms
D:\教材\数据结构\交付学生实验素材\Heap>Heap.exe num.txt
Heapsize(K)：8
运行时间：364.329ms
```

```
D:\教材\数据结构\交付学生实验素材\Heap>Heap.exe num.txt
Heapsize(K)：9
运行时间：374.749ms
D:\教材\数据结构\交付学生实验素材\Heap>Heap.exe num.txt
Heapsize(K)：10
运行时间：0.8269ms

D:\教材\数据结构\交付学生实验素材\Heap>Heap.exe num.txt
Heapsize(K)：10
运行时间：404.143ms

D:\教材\数据结构\交付学生实验素材\Heap>Heap.exe num.txt
Heapsize(K)：15
运行时间：429.062ms

D:\教材\数据结构\交付学生实验素材\Heap>Heap.exe num.txt
Heapsize(K)：20
运行时间：434.783ms
```

图 4　原始数据记录

以 In（K）作为横坐标，时间 t 为纵坐标，使用 matlab 绘制散点图如图 5 所示，在 N 值确定的时候，发现时间 t 与 k 的对数基本构成线性关系，但由于未知的因素，使得实验有一定误差。在误差允许的范围内，耗时与所构造的小顶堆的深度成正比。

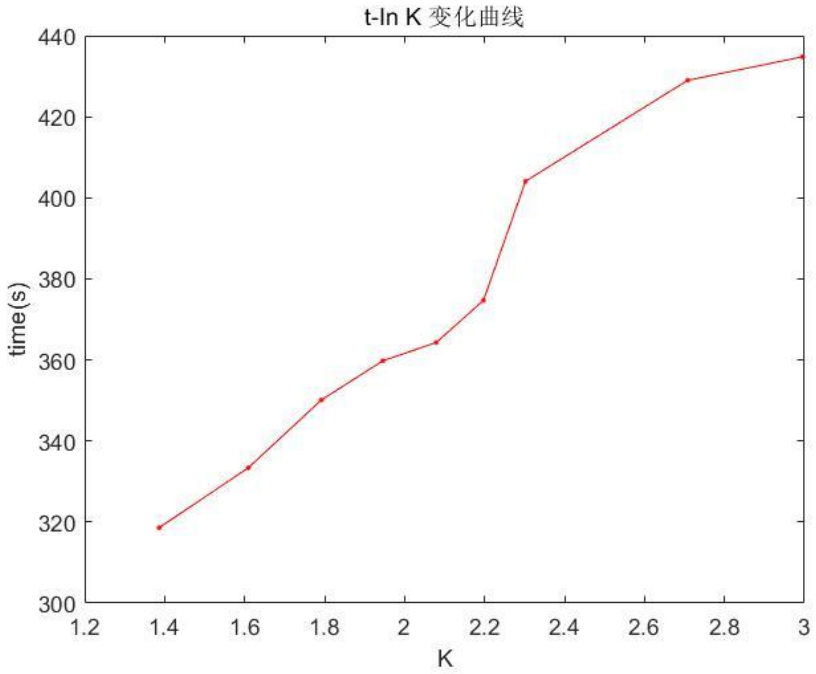


图 5　t 随 In（K）的变化曲线(N=100)

3.探究随机数组成与运行时间的关系：

实验过程中我发现，相同的 N 和 K，不同的随机数组合，运行时间会有一

定差别，添加实验验证。控制数据总量 N=100，范围[0,100)，K=10，实验 python 生成 10 组不同的随机数，比较运行时间，结果如下图 6。

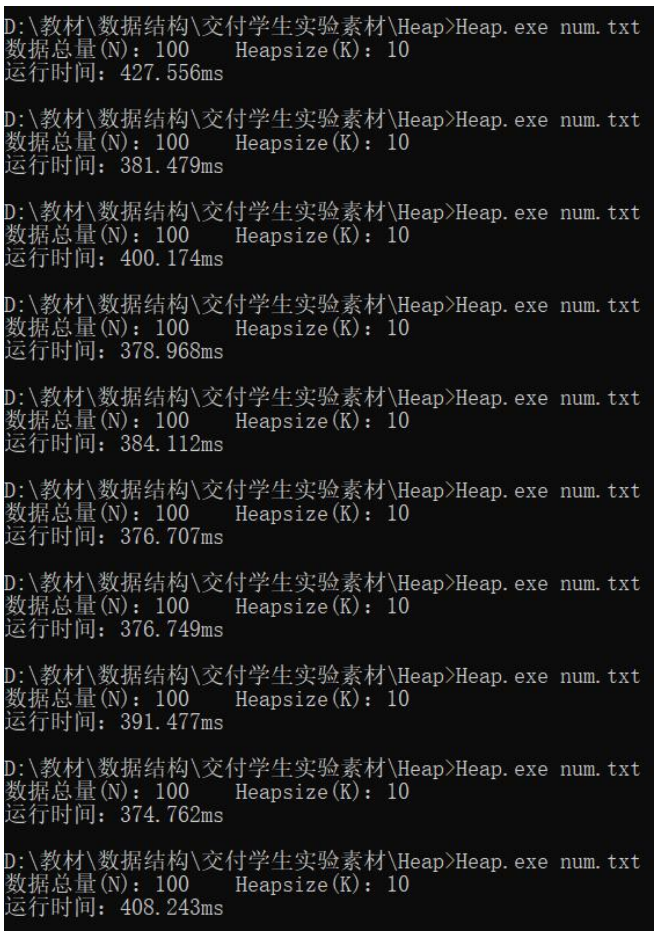


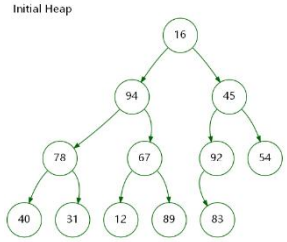
图 6　不同随机数运行结果

分析上图 6 可看出，随机数的组成会一定程度上影响程序运行时间，在 N=100，范围[0,100)，K=10 的条件下，不同随机数运行时间极差为 52.79ms，已达到最短运行时间（376.707ms）的 14%。

4.构造图像

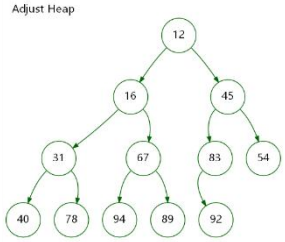
在测试中，构造了 100 个[0，100)范围内的测试数据：16 94 45 78 67 92 54 40 31 12 89 83 52 93 33 73 79 91 75 23 44 27 35 57 2 32 10 47 65 1 99 5 18 80 96 48 20 34 71 76 75 55 75 63 8 46 41 4 24 33 11 18 9 98 78 65 3 71 51 21 55 72 51 54 61 23 24 3 21 17 99 36 58 57 32 36 1 46 6 42 73 22 62 8 34 91 61 46 9 28 45 56 2 1 67 55 79 14 66 30，K 值取 12，构造了堆调整的每个过程，如图 7 所示。

Initial Heap



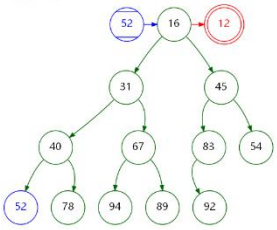
value		16	94	45	78	67	92	54	40	31	12	89	83
address	0	1	2	3	4	5	6	7	8	9	10	11	12

Adjust Heap



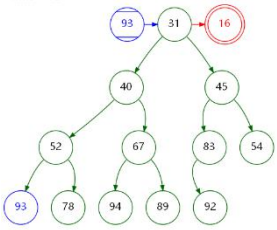
value		12	16	45	31	67	83	54	40	78	94	89	92
address	0	1	2	3	4	5	6	7	8	9	10	11	12

Time = 11



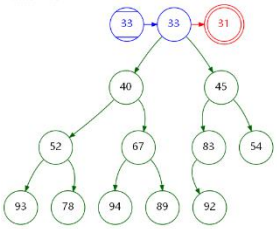
value		16	31	45	40	67	83	54	52	78	94	89	92
address	0	1	2	3	4	5	6	7	8	9	10	11	12

Time = 12



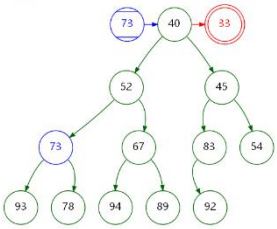
value		31	40	45	52	67	83	54	93	78	94	89	92
address	0	1	2	3	4	5	6	7	8	9	10	11	12

Time = 13



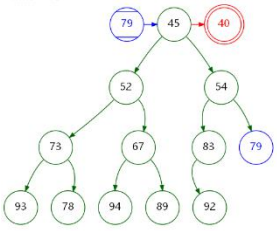
value		33	40	45	52	67	83	54	93	78	94	89	92
address	0	1	2	3	4	5	6	7	8	9	10	11	12

Time = 14



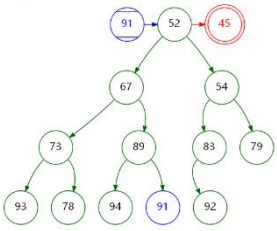
value		40	52	45	73	67	83	54	93	78	94	89	92
address	0	1	2	3	4	5	6	7	8	9	10	11	12

Time = 15



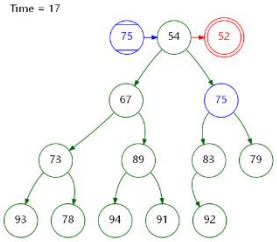
value		45	52	54	73	67	83	79	93	78	94	89	92
address	0	1	2	3	4	5	6	7	8	9	10	11	12

Time = 16



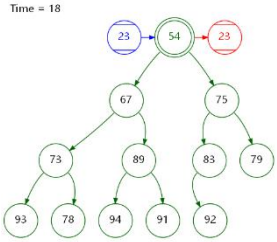
value		52	67	54	73	89	83	79	93	78	94	91	92
address	0	1	2	3	4	5	6	7	8	9	10	11	12

Time = 17



value		54	67	75	73	89	83	79	93	78	94	91	92
address	0	1	2	3	4	5	6	7	8	9	10	11	12

Time = 18



value		54	67	75	73	89	83	79	93	78	94	91	92
address	0	1	2	3	4	5	6	7	8	9	10	11	12

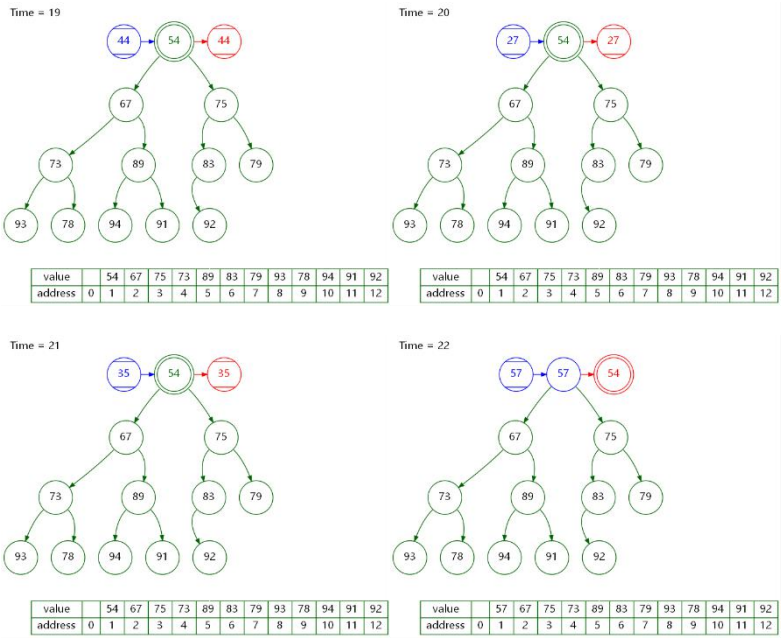


图 7　一次 Top12 的小顶堆调整过程

八、总结及心得体会：

使用堆的形式维护数组可以降低查找的时间复杂度（类似二分查找），同时并没有大幅度牺牲了空间复杂度，消耗空间较小，并且有着较高的稳定性，即最坏时间复杂度与平均时间复杂度相等。

但同时，堆的维护需要消耗时间，有着较高的维护成本。

本次实验让我学习到了算法设计时，应该关注我们要的结果（本实验中的 K），从结果出发，尽量简化问题，可以优化算法。

在实验过程中，我发现了该算法的运行时间不光与 N，K 等参数有关，与输入随机数的构成也有关，算法具有一定的随机性。具体表现为在 N，K 不变的情况下，生成不同的输入随机数组合会影响算法的运行时间；同时，完全相同的输入数据与参数，每次运行的时间也不尽相同。

九、对本实验过程及方法、手段的改进建议：

1. 实验过程中发现随机数的组成对运行时间有一定影响，可以添加随机数生成程序，便于实验比较

2. 添加顺序存储的代码，与以堆形式存储的进行对比，加深理解

3. 添加计算程序运行时间的代码，方便探究运行时间与 N，K 的关系

电 子 科 技 大 学

实 验 报 告

实验三

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：

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，使用了广度优先搜索解决赋权有向图或者无向图的单源最短路径问题。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。

Dijkstra 算法维护一个 dist 数组记录最短距离，同时维护一个 path 数组记录最短路径，不断更新数组直到 T 集合为空。所有没有直接相连的点之间的距离初始化为无穷。

五、实验器材（设备、元器件）

硬件平台：处理器: Intel(R) Core(TM) i7-9750H CPU @2.60GHz (12 CPUs), ~2592MHz 内存:16384MB RAM 外存：512GB SSD 显卡：NVIDIA GeForce GTX 1660 Ti

软件平台：操作系统: Windows 10 家庭中文版 64 位(10.0，内部版本 19043)

IDE：Dev-C++ 5.11.0.0 编译器：gcc version 8.1.0

六、实验步骤

算法伪代码如下图 1：

Algorithm : Dijkstra

Input : Directed graph $G = (V, E, W)$ with weight

Output : All the shortest paths from the source vertex s to every vertex $v_i \in V \setminus \{s\}$

$1: S \leftarrow \{s\}$
$2: \text{dist}[s, s] \leftarrow 0$
$3:$ **for** $v_i \in V - \{s\}$ **do**
$4:$ $\quad \text{dist}[s, v_i] \leftarrow w(s, v_i)$
$\quad (\text{when } v_i \text{ not found}, \text{dist}[s, v_i] \leftarrow \infty)$
$5:$ **while** $V - S \neq \varnothing$ **do**
$6:$ $\quad$ find $\min_{v_j \in V} \text{dist}[s, v_i]$ from the set $V - S$
$7:$ $\quad S \leftarrow S \cup \{v_j\}$
$8:$ $\quad$ **for** $v_i \in V - S$ **do**
$9:$ $\quad\quad$ **if** $\text{dist}[s, v_j] + w_{j,i} < \text{dist}[s, v_i]$ **then**
$10:$ $\quad\quad\quad \text{dist}[s, v_i] \leftarrow \text{dist}[s, v_j] + w_{j,i}$

(10)

图 1 Dijkstra 算法伪代码

算法的流程如下：

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。

实验中添加了代码“srand((unsigned)time(0));”以使每次生成的随机数不同。

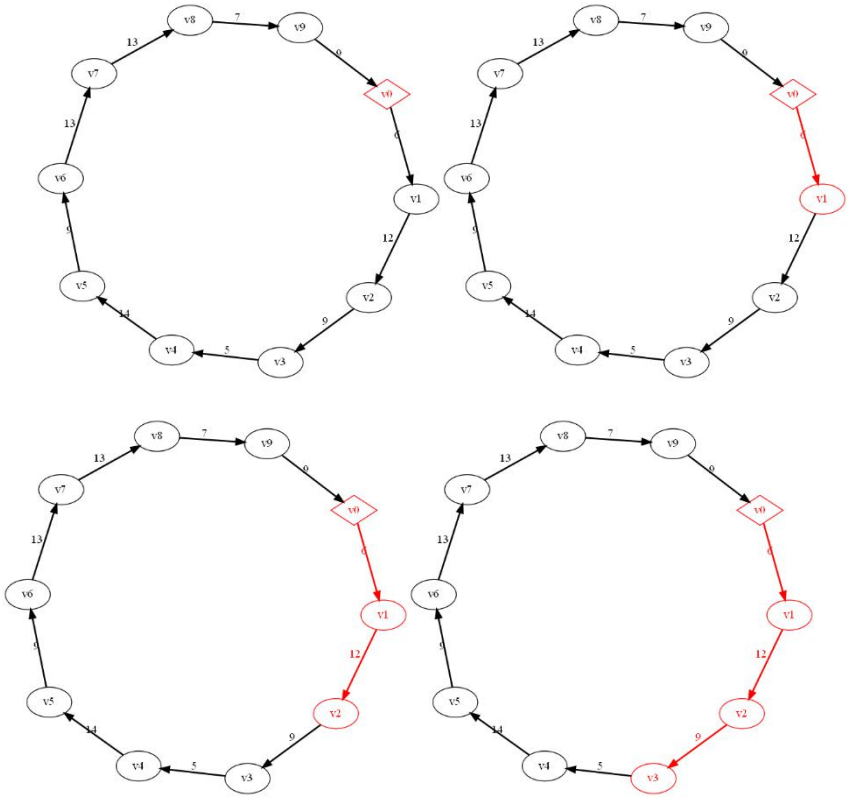
本次实验过程中不断改变节点数 node 和连边数 edge 并绘图展示，学习 Dijkstra 算法的运行过程，加深对概念的理解。

七、实验数据及结果分析

节点初始为黑色，起始点 V0 为红色，此后每执行一次将新节点加入

固定 node=10，改变连边数，观察结果。

当 edge<10 时，产生的图中初始节点 v0 不与其它节点连通，不会更新。一种特殊情况是 edge=node，此时会形成环，得到的结果如下图 2。



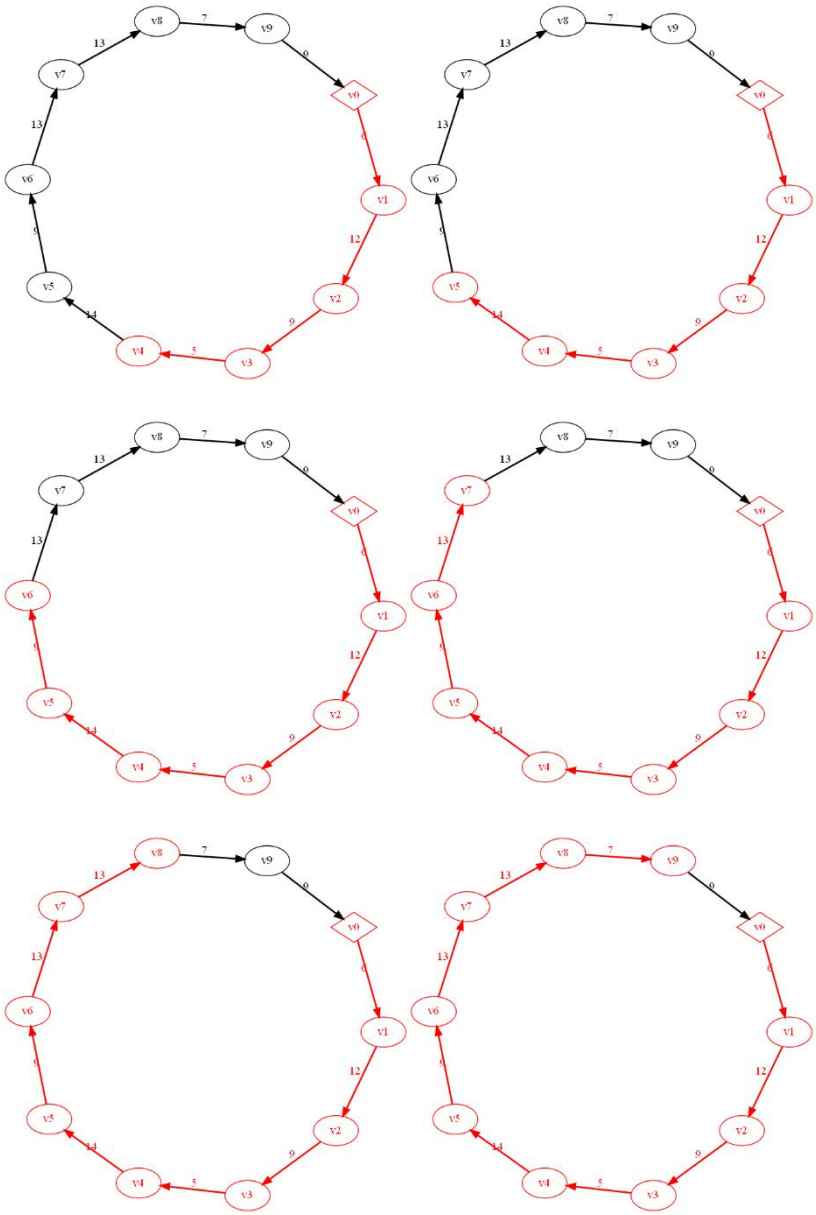
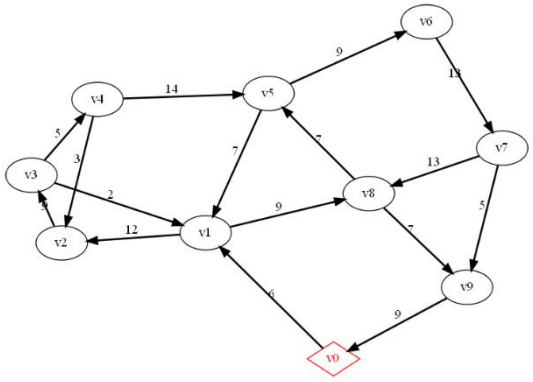
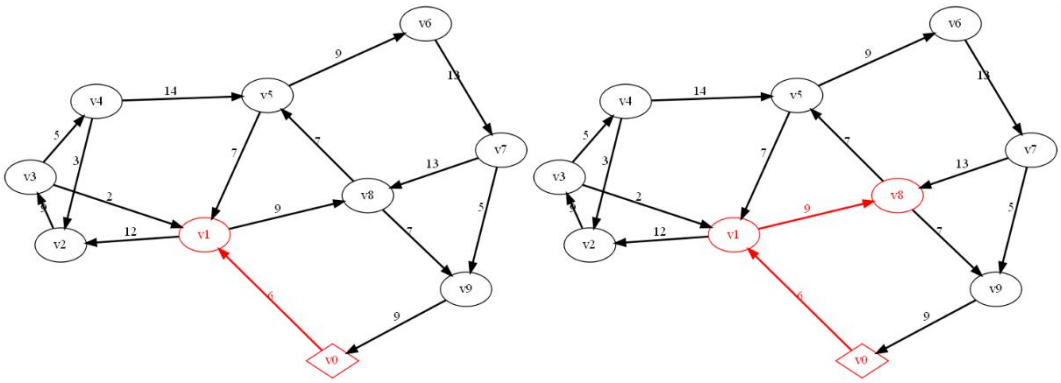
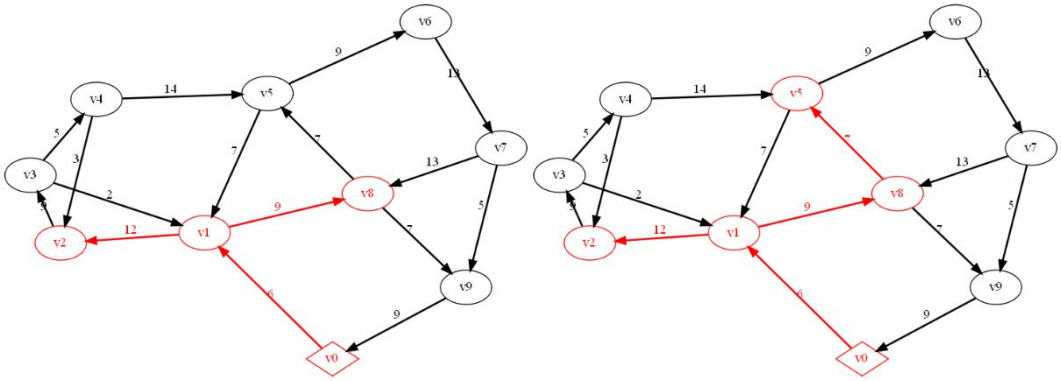
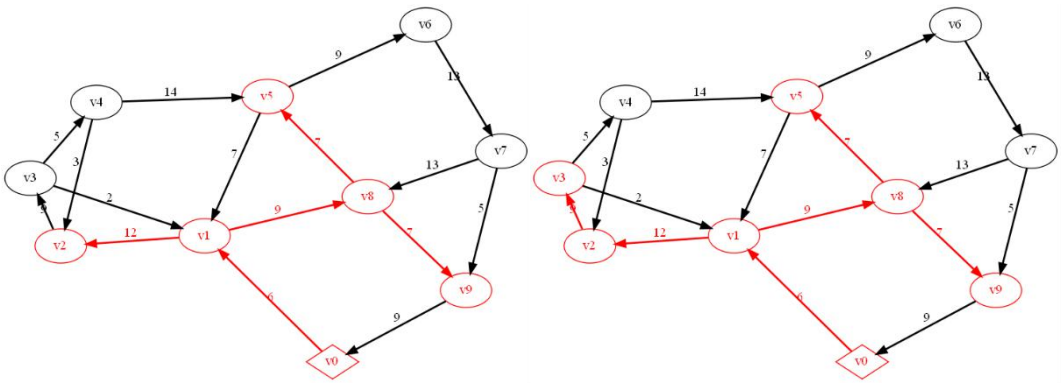
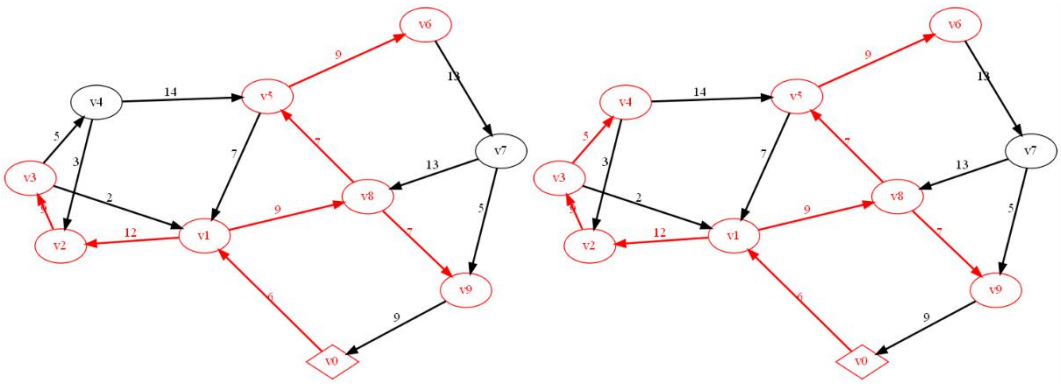


图 2　一次 10 个节点 10 条边有向图最短路径算法过程

令 edge=16 得到的一般情况如下图 3。



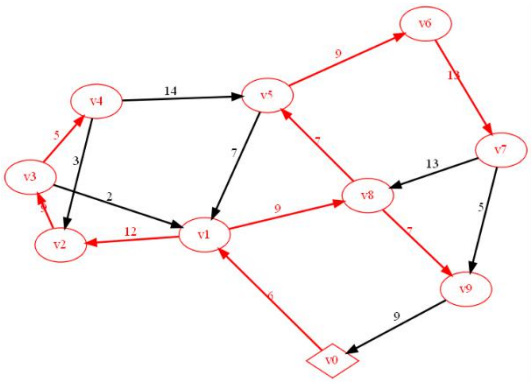
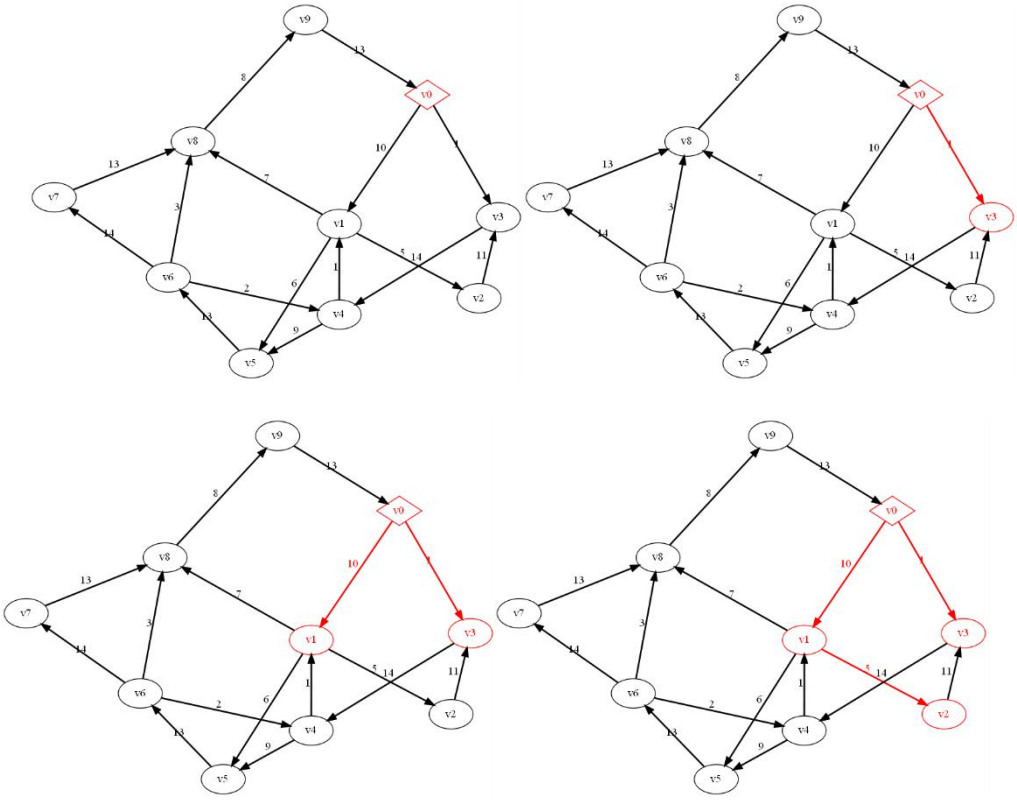


图 3　一次 10 个节点 16 条边有向图最短路径算法过程

实验过程中我发现了当 node，edge 两个参数一定时，每次运行生成的结果都是一样的，添加了代码“srand((unsigned)time(0));”，使得 rand 生成的随机数不固定。保持 node=10，edge=16，修改后再运行代码得到下图 4。



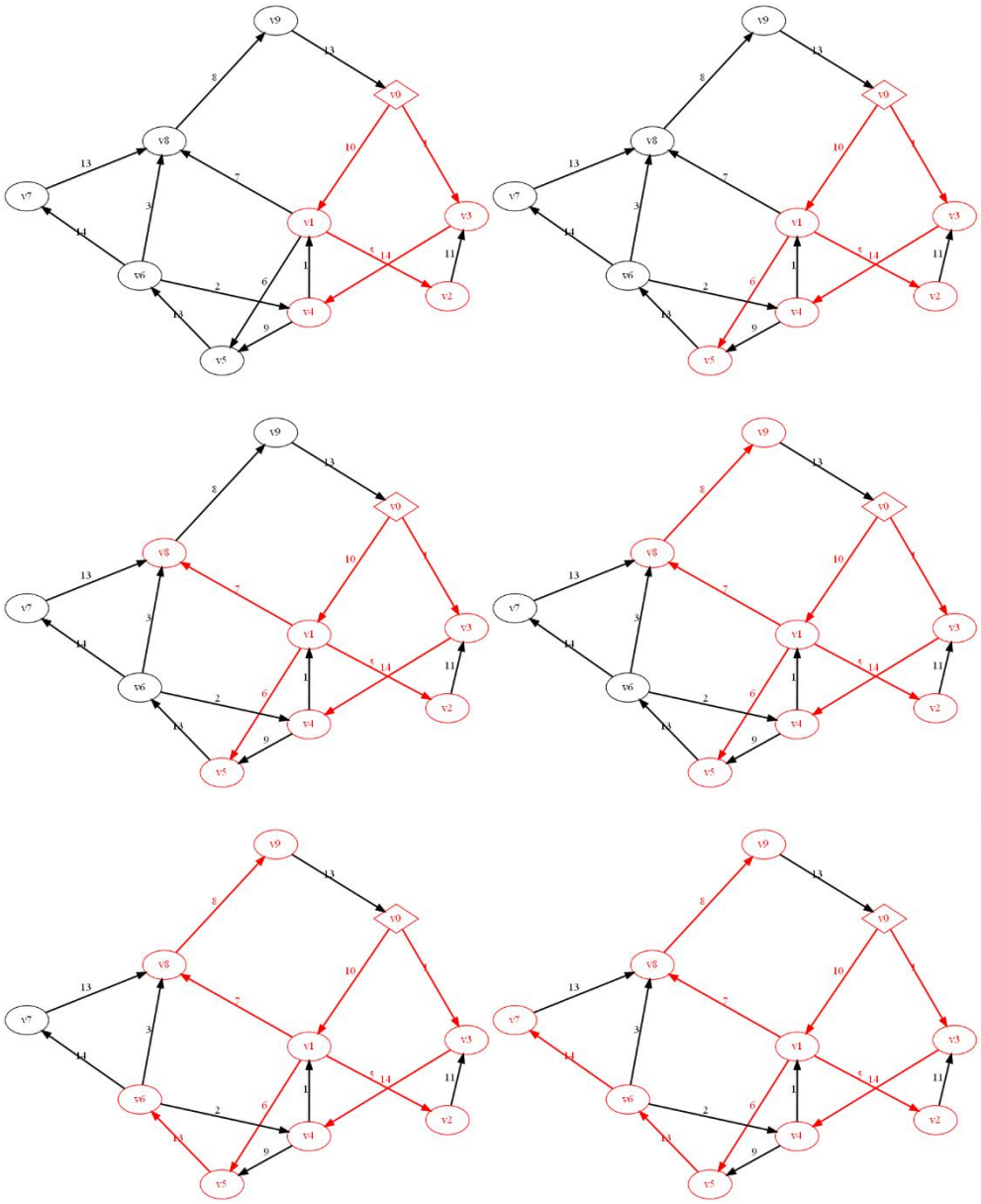


图 4　一次 10 个节点 16 条边有向图最短路径算法过程

分析上述三幅图，可以清楚地看出 Dijkstra 算法每一步的更新过程，帮助我们理解。

八、总结及心得体会：

Dijkstra 算法能够有效地解决单源最短路径问题，其时间复杂度为 $O(n^2)$。并且算法简单，易于实现。但是 Dijkstra 算法不能处理负权重的情况，有一定的局限性。对比其与 Floyd 算法，Dijkstra 算法是单源的，只能算出一个节点到其它节点的最短距离；而 Floyd 算法是全源的，可以求出任意两节点之间的最短路径，同时可以处理负权重的情况（无负回路即可）。

九、对本实验过程及方法、手段的改进建议：

1. Dijkstra 算法可以使用堆来优化，将其时间复杂度降为 O(nlogn)。可以添加相应代码，以便于学生更深刻地理解 Dijkstra 算法与堆的概念及应用。

2.添加 Floyd 算法对比