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：黄谨楠

学　　号：**2016110902005**

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) 在二叉排序树中插入节点

二叉排序树的结构通常不是一次生成的，而是在查找过程中，当树中不存在关键字等于给定值的结点时在进行插入，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。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

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) 对二叉排序树进行中序遍历

若二叉排序树为空，遍历结束，否则：中序遍历根结点的左子树；访问根结点；中序遍历根结点的右子树。

(4) 二叉排序树的查找

若要在二叉排序树中查找关键字值为 key 的结点，根据二叉排序树的特点，其查找过程为：

1. 若二叉排序树为空树，查找失败。

2. 若二叉排序树非空，比较 key 与根结点关键字值的大小，若相等，查找成功，否则：

①若 key 小于根结点的关键字值，转向 1.在根结点的左树上继续查找。

②若 key 大于根结点的关键字值，转向 1.在根结点的右树上继续查找。

平衡二叉树的实现

(1)构造平衡二叉树(包括在平衡二叉树中插入结点，以及对二叉排序树进行调整)

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。

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。

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。

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。

(2) 对平衡二叉树 T 作中序遍历

若平衡二叉树为空，遍历结束，否则：中序遍历根结点的左子树；访问根结点；中序遍历根结点的右子树。

五、实验器材（设备、元器件）

硬件平台

CPU：intel CORE i7-8750H

内存：DDR4 2666MHz 16G

GPU：NVIDIA GEFORCE GTX1070

硬盘：M.2 固态 512G + SSHD 1T

软件平台

操作系统：WIN10 1803

开发环境：C-Free 5, Graphviz 2.39

测试环境：mingw5

六、实验步骤

二叉排序树：

(1) 用二叉链表作存储结构，生成一棵二叉排序树 T。

```
void  CreateBST(BSTree  *bst, char * filename)
/*从文件输入元素的值，创建相应的二叉排序树*/
{
    FILE *fp;
    KeyType keynumber;
    *bst=NULL;
    fp = fopen(filename,"r+");
    if(fp==NULL)
        exit(0x01);
    while(EOF != fscanf(fp,"%d",&keynumber))
        InsertBST(bst, keynumber);
}
```

(2) 对二叉排序树 T 作中序遍历，输出结果。

```
void InOrderCleanFlag(BSTree root)
/*中序遍历二叉树，root 为指向二叉树根结点的指针*/
{
    if(root != NULL)
    {
        InOrderCleanFlag(root->lchild); /*中序遍历左子树*/
        root->flag = 0; /*输出结点*/
        printf("%d ",root->key);
        InOrderCleanFlag(root->rchild); /*中序遍历右子树*/
    }
}
```

(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。

```
BSTree  SearchBST(BSTree bst, KeyType key)
/*在根指针 bst 所指二叉排序树 bst 上，查找关键字等于 key 的结点，若查找成功，返回指向该
元素结点指针，否则返回空指针*/
{
    BSTree q;
    q=bst;
    while(q)
    {
```

```
        q->flag=1;
        if (q->key == key)
        {
            q->flag=2;
            return q;  /*查找成功*/
        }
        if (q->key > key)
            q=q->lchild;  /*在左子树中查找*/
        else
            q=q->rchild;  /*在右子树中查找*/
    }
    return NULL; /*查找失败*/
}/*SearchBST*/
```

二叉平衡树：

(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。

在平衡二叉树中进行插入的函数：

```
void  ins_AVLtree(AVLTree  *avlt ,  KeyType  K)
/*在平衡二叉树中插入元素 K，使之成为一棵新的平衡二叉排序树*/
{
    AVLTNode *S;
    AVLTNode *A,*FA,*p,*fp,*B,*C;
    S=(AVLTree)malloc(sizeof(AVLTNode));
    S->key=K;
    S->lchild=S->rchild=NULL;
    S->bf=0;
    if (*avlt==NULL)
        *avlt=S;
    else
    {
    /* 首先查找 S 的插入位置 fp，同时记录距 S 的插入位置最近且
        平衡因子不等于 0（等于-1 或 1）的结点 A，A 为可能的失衡结点*/
        A=*avlt;  FA=NULL;
        p=*avlt;  fp=NULL;
        while  (p!=NULL)
        {
            if (p->bf!=0)
            {
                A=p; FA =fp;
            }
            fp=p;
            if  (K < p->key)
                p=p->lchild;
            else if (K > p->key)
                p=p->rchild;
            else
            {
                free(S);
                return;
            }
        }
        /* 插入 S*/
        if (K < fp->key)
```

```
        fp->lchild=S;
    else
        fp->rchild=S;
    /* 确定结点 B，并修改 A 的平衡因子 */
    if (K < A->key)
    {
        B=A->lchild;
        A->bf=A->bf+1;
    }
    else
    {
        B=A->rchild;
        A->bf=A->bf-1;
    }
    /* 修改 B 到 S 路径上各结点的平衡因子（原值均为 0）*/
    p=B;
    while (p!=S)
        if  (K < p->key)
        {
            p->bf=1;
            p=p->lchild;
        }
        else
        {
            p->bf=-1;
            p=p->rchild;
        }
    /* 判断失衡类型并做相应处理 */
    if  (A->bf==2 && B->bf==1)       /* LL 型 */
    {
        B=A->lchild;
        A->lchild=B->rchild;
        B->rchild=A;
        A->bf=0;
        B->bf=0;
        if (FA==NULL)
            *avlt=B;
        else
            if (A==FA->lchild)
                FA->lchild=B;
            else
                FA->rchild=B;
    }
    else
        if (A->bf==2 && B->bf==-1)       /* LR 型 */
        {
            B=A->lchild;
            C=B->rchild;
            B->rchild=C->lchild;
            A->lchild=C->rchild;
            C->lchild=B;
            C->rchild=A;
            if (S->key < C->key)
            {
                A->bf=-1;
                B->bf=0;
                C->bf=0;
```

```
            }
            else
                if (S->key >C->key)
                {
                    A->bf=0;
                    B->bf=1;
                    C->bf=0;
                }
                else
                {
                    A->bf=0;
                    B->bf=0;
                }
                if  (FA==NULL)
                    *avlt=C;
                else
                    if (A==FA->lchild)
                        FA->lchild=C;
                    else
                        FA->rchild=C;
        }
        else
            if  (A->bf==-2 && B->bf==1)      /* RL型 */
            {
                B=A->rchild;
                C=B->lchild;
                B->lchild=C->rchild;
                A->rchild=C->lchild;
                C->lchild=A;
                C->rchild=B;
                if (S->key <C->key)
                {
                    A->bf=0;
                    B->bf=-1;
                    C->bf=0;
                }
                else
                    if (S->key >C->key)
                    {
                        A->bf=1;
                        B->bf=0;
                        C->bf=0;
                    }
                    else
                    {
                        A->bf=0;
                        B->bf=0;
                    }
                    if (FA==NULL)
                        *avlt=C;
                    else
                        if (A==FA->lchild)
                            FA->lchild=C;
                        else
                            FA->rchild=C;
            }
            else
```

```
                if (A->bf==-2 && B->bf==-1)      /* RR型 */
                {
                    B=A->rchild;
                    A->rchild=B->lchild;
                    B->lchild=A;
                    A->bf=0;
                    B->bf=0;
                    if (FA==NULL)
                        *avlt=B;
                    else
                        if (A==FA->lchild)
                            FA->lchild=B;
                        else
                            FA->rchild=B;
                }
    }
}
```

创建一棵平衡二叉树(从 txt 文件中读取数据，调用 ins_AVLtree 函数):

```
void  CreateAVLT(AVLTree  *bst, char * filename)
/*从文件输入元素的值，创建相应的二叉排序树*/
{
    FILE *fp;
    KeyType keynumber;
    *bst=NULL;
    fp = fopen(filename,"r+");
    while(EOF != fscanf(fp,"%d",&keynumber))
        ins_AVLtree(bst, keynumber);
}
```

(2) 对平衡二叉树 T 作中序遍历，输出结果。

```
void InOrder(AVLTree root)
//中序遍历平衡二叉树，root 为指向该平衡二叉树根结点的指针
{
    if(root!=NULL)
    {
        InOrder(root->lchild);
        printf("%d(%d)\t",root->key,root->bf);
        InOrder(root->rchild);
    }
}
```

七、实验数据及结果分析

二叉排序树、二叉平衡树实验数据及结果分析:

生成测试数据:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <iostream>
using namespace std;
#define N 1000

int main()
{
    int s[N];
    int i=0;
    FILE *fp;
    fp = fopen("F:\\CFreeProj\\BST\\Num.txt","a");
    for(i=0; i<=99 ;i++)
    {
        s[i] = rand()%1000;
        fprintf(fp,"%d  ",s[i]);
        //printf("%d  ",s[i]);
        if(s[i] == -1)
            s[i] = 0;
    }
    fclose(fp);
    return 0;
}
```

在测试中，构造了 100 个取值在 1000 以内的数据元素序列，并将其存入一个 txt 文件中，命名为 Num.txt: 41 467 334 500 169 724 478 358 962 464 705 145 281 827 961 491 995 942 827 436 391 604 902 153 292 382 421 716 718 895 447 726 771 538 869 912 667 299 35 894 703 811 322 333 673 664 141 711 253 868 547 644 662 757 37 859 723 741 529 778 316 35 190 842 288 106 40 942 264 648 446 805 890 729 370 350 6 101 393 548 629 623 84 954 756 840 966 376 931 308 944 439 626 323 537 538 118 82 929 541。

得到的排序二叉树如图 1 所示：

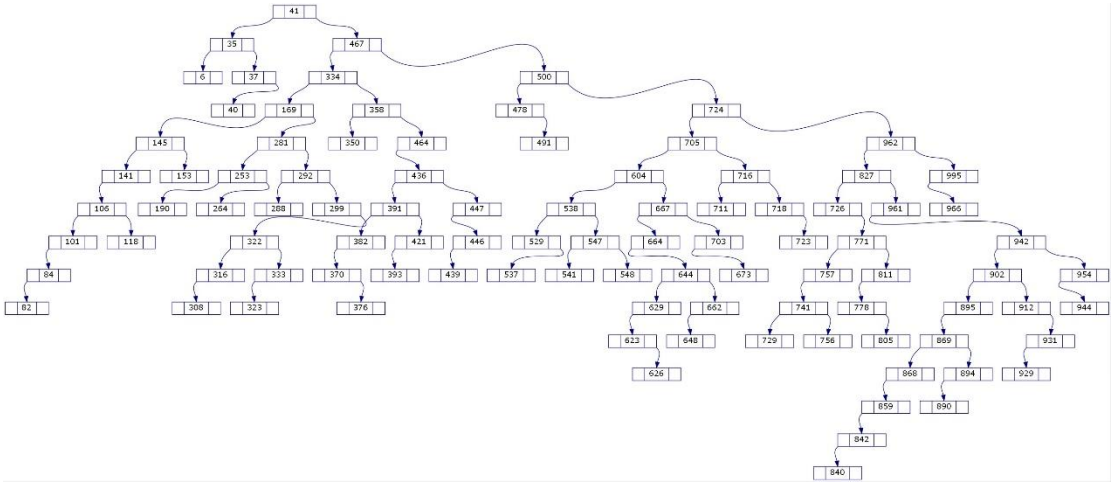

图 1-排序二叉树生成结果图

(2) 二叉排序树的中序遍历是一个递增的有序序列，对该二叉排序树作中序遍历，输出结果如图 2 所示：

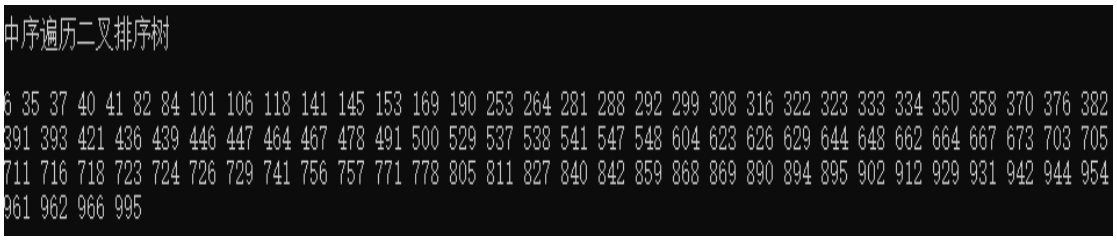

图 2-中序遍历二叉排序树

(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。首先输入元素 393，查找二叉排序树 T，查命中，结果如图 3 所示：

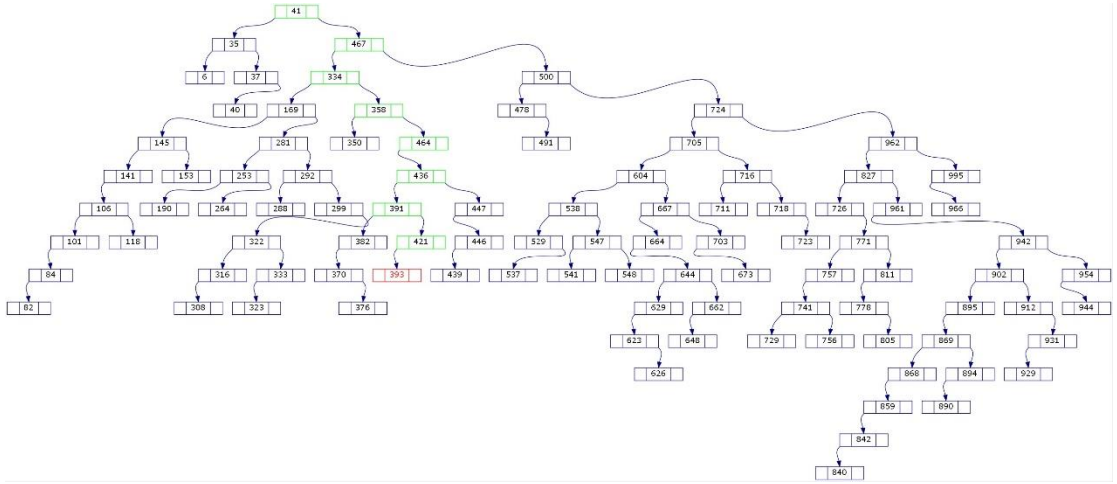

图 3-查找元素 393

输入元素 840，查命中，输出结果如图 4 所示：

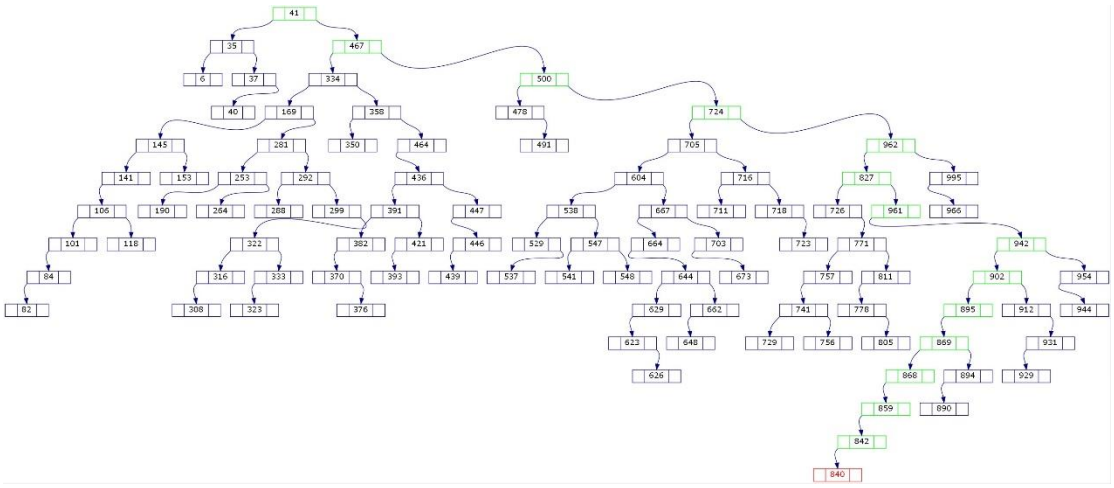

图 4-查找元素 840

输入元素 555，查不命中，输出结果如图 5 所示：

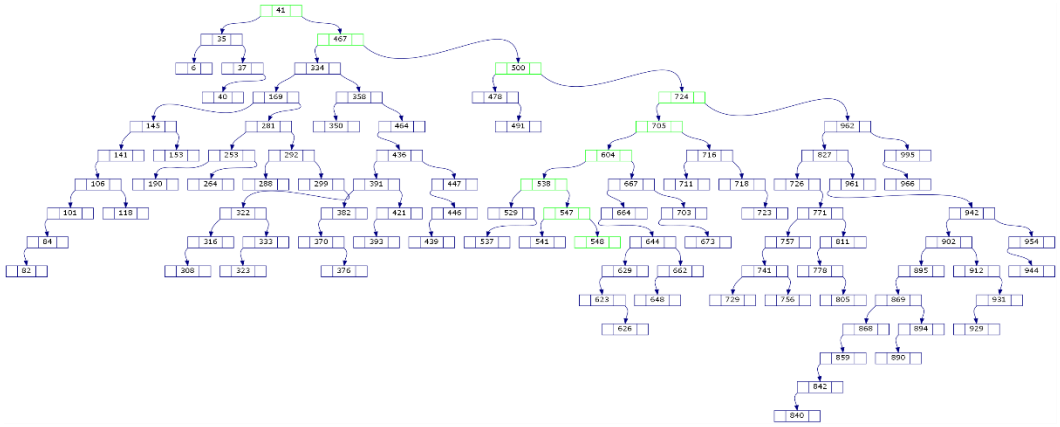

图 5-查找元素 555

对于同样的输入序列，生成得到的平衡二叉树如图 6 所示。

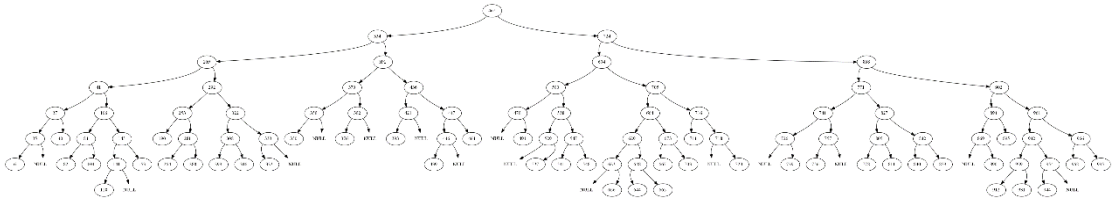

图 6-平衡二叉树生成结果图

对该平衡二叉树进行中序遍历得到的结果如图 7 所示，结果同样是一个递增的有序序列，其中括号中的数为以该结点为根的二叉树中左子树与右子树高度之差。

```
中序遍历平衡二叉树

6(0)    35(1)   37(1)   40(0)   41(-1)  82(0)   84(0)   101(0)  106(-1) 118(0)  141(1)  145(1)  153(0)  169(1)  190(0)
253(-1) 264(0)  281(0)  288(0)  292(0)  299(0)  308(0)  316(0)  322(0)  323(0)  333(1)  334(1)  350(0)  358(1)  370(0)
376(0)  382(1)  391(-1) 393(0)  421(1)  436(-1) 439(0)  446(1)  447(1)  464(0)  467(0)  478(-1) 491(0)  500(-1) 529(-1)
537(0)  538(0)  541(0)  547(0)  548(0)  604(-1) 623(-1) 626(0)  629(0)  644(0)  648(0)  662(0)  664(1)  667(0)  673(0)
703(0)  705(1)  711(0)  716(-1) 718(-1) 723(0)  724(0)  726(-1) 729(0)  741(0)  756(0)  757(1)  771(0)  778(0)  805(0)
811(0)  827(0)  840(0)  842(0)  859(0)  868(-1) 869(-1) 890(0)  894(1)  895(0)  902(-1) 912(0)  929(0)  931(0)  942(0)
944(0)  954(1)  961(1)  962(0)  966(0)  995(0)
```

图 7-中序遍历平衡二叉树

平衡二叉树在不是空树的情况下是一种特殊的二叉排序树，它具有如下性质：它的左右子树都是平衡二叉树，并且左右子树的深度之差不超过 1，即每个左、右子树深度之差不超过 1。通过对比图 5 和图 1，可以发现平衡二叉树的深度明显小于普通排序二叉树的深度。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。

八、总结及心得体会：

二叉排序树有以下特点：(1)若左子树不空，则左子树上所有结点的值均小于

它的根结点的值；(2)若右子树不空，则右子树上所有结点的值均大于它的根结点的值；(3)左、右子树也分别为二叉排序树；(4)没有键值相等的节点。这意味着二叉排序树在存储数据的方式上应用了二分法的思想，从而减小了在二叉排序树上进行查找时的时间复杂度。

但二叉排序树也有缺点：在极端情况下，二叉排序树可能只存在左子树或者右子树。这使得二叉排序树的深度非常深，在这种情况下的二叉排序树上进行查询时间复杂度会增加。

平衡二叉树是一种特殊的二叉排序树，是对一般二叉排序树的一种改进。最大程度上降低了二叉排序树的深度，保证了进行查询时较低的时间复杂度。

九、对本实验过程及方法、手段的改进建议：

1.可以加入二叉排序树结点的删除，使得删除之后的二叉排序树仍是二叉排序树。

2.可以加入平衡二叉树的调整过程。

3.测试数据的选择可以更加多样化，除了随机数，可以选择其他字符。

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

堆排序的思想：利用大顶堆(小顶堆)堆顶记录的是最大关键字(最小关键字)这一特性，使得每次从无序中选择最大记录(最小记录)变得简单。

其基本思想为(大顶堆)：

1)将初始待排序关键字序列(R1,R2....Rn)构建成大顶堆，此堆为初始的无序区；

2)将堆顶元素 R[1]与最后一个元素 R[n]交换，此时得到新的无序区(R1,R2,.....Rn-1)和新的有序区(Rn),且满足 R[1,2...n-1]<=R[n];

3)由于交换后新的堆顶 R[1]可能违反堆的性质，因此需要对当前无序区(R1,R2,.....Rn-1)调整为新堆，然后再次将 R[1]与无序区最后一个元素交换，得到新的无序区(R1,R2....Rn-2)和新的有序区(Rn-1,Rn)。不断重复此过程直到有序区的元素个数为 n-1，则整个排序过程完成。

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

CPU：intel CORE i7-8750H

内存：DDR4 2666MHz 16G

GPU：NVIDIA GEFORCE GTX1070

硬盘：M.2 固态 512G + SSHD 1T

软件平台

操作系统：WIN10 1803

开发环境：C-Free 5, Graphviz 2.39

测试环境：Mingw5

六、实验步骤

(1) 构造堆，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：

```
void HeapAdjust(int array[],int i,int nLength)
{
    int nChild;
    int nTemp;
    for(;2*i+1<nLength;i=nChild)
    {
        nChild=2*i+1;
        if(nChild<nLength-1&&array[nChild+1]<array[nChild])
            ++nChild;
        if(array[i]>array[nChild])
        {
            nTemp=array[i];
            array[i]=array[nChild];
            array[nChild]=nTemp;
        }
        else break;
    }
}
```

(2) 实现 Top-K 问题求解的过程：

```
int main(int argc, char *argv[])
{
    if(argc<3)
        printf("Heap.exe [N] [K]\n");

    char figlabel[128];
    char orderstr[128];
    int Nnum = atoi(argv[1]);
    int Heapsize = atoi(argv[2]);
    int Heap[Heapsize];
    memset(Heap,'\0',sizeof(Heap));

////////////////////////////////////////////
    srand((unsigned)time(0));
    for(int i=0; i<Heapsize; i++)
        Heap[i] = rand();

    sprintf(figlabel, "Initial Heap");
```

```
    DotHeap(Heap,Heapsize,figlabel);
    sprintf(orderstr, "dot.exe -Tpng heapT.gv -o IniT.png");
    system(orderstr);

    for(int i=Heapsize/2; i>=0; --i)
        HeapAdjust(Heap,i,Heapsize);
    sprintf(figlabel, "Adjust Heap");
    DotHeap(Heap,Heapsize,figlabel);
    sprintf(orderstr, "dot.exe -Tpng heapT.gv -o AdjT.png");
    system(orderstr);

    for(int i = Heapsize; i<RAND_MAX;i++)
    {
        int temp = rand();
        if(temp>Heap[0])
        {
            int pretop = Heap[0];
            Heap[0] = temp;
            HeapAdjust(Heap,0,Heapsize);
        }
    }

    sprintf(figlabel, "Initial Heap");
    DotHeap(Heap,Heapsize,figlabel);
    sprintf(orderstr, "dot.exe -Tpng heapT.gv -o FinalT.png");
    system(orderstr);
    return 0;
}
```

七、实验数据及结果分析

选取 k=1024 的一列，以 e 作为横坐标，log2（t）为纵坐标，绘制散点图 1 所示。

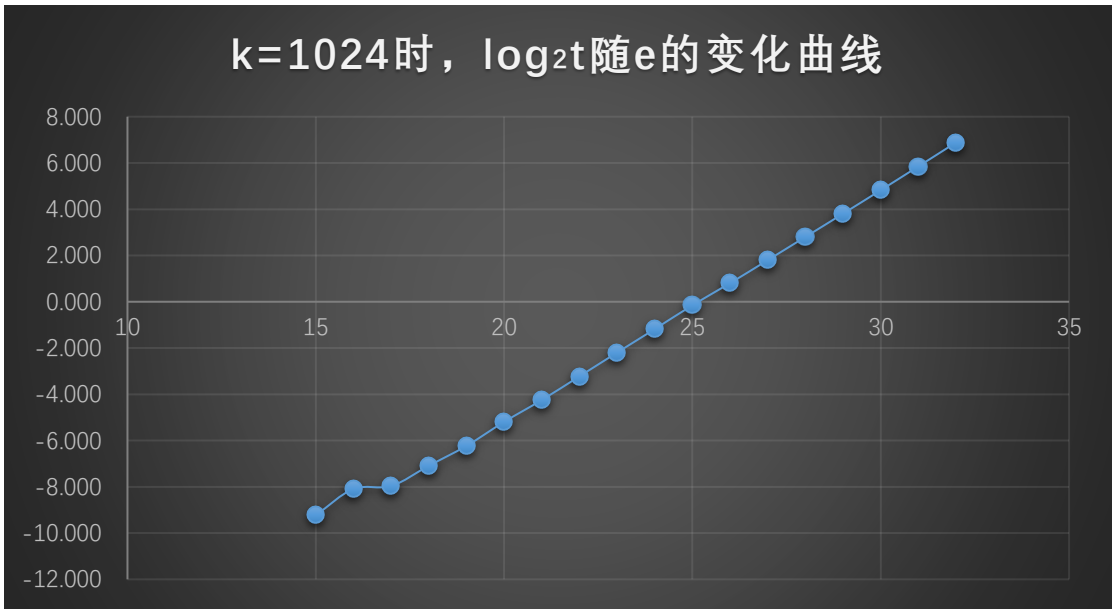

图 1 k=1024 时，$\log_2 t$ 随 e 的变化曲线

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

以 e 作为横坐标，log2（t）为纵坐标，绘制散点图如图 2 所示，在 e 值确定的时候，时间 t 与 k 的对数约成线性关系，但由于未知的因素，使得实验误差较大。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。

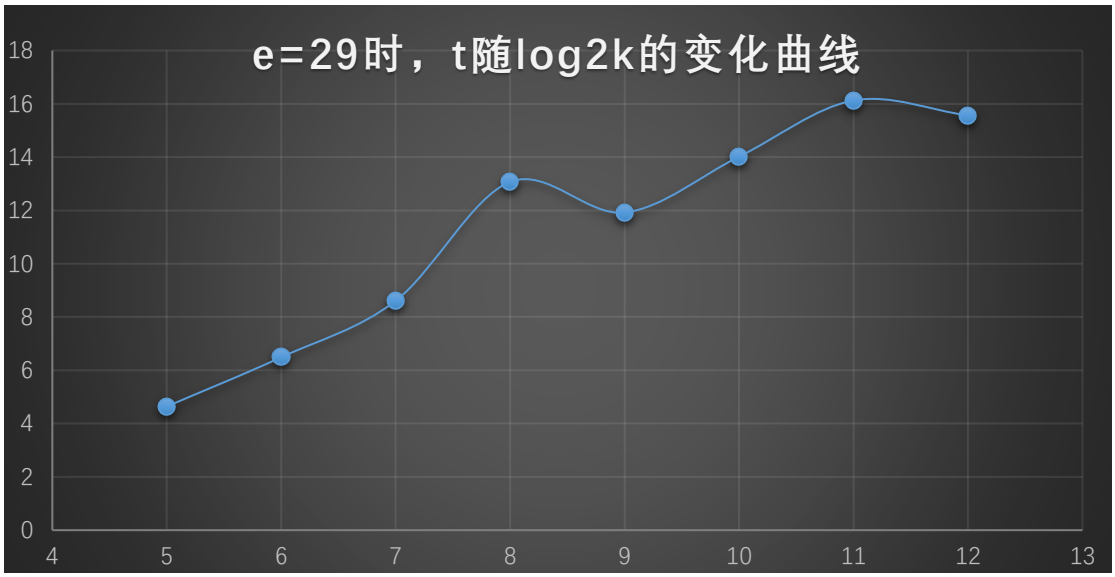

图 2　e=29 时，t 随 log2k 的变化曲线

使用.bat 脚本生成测试数据(5000 个随机数中选取前 15 个):

```
@echo off
call Clean.bat
Heap.exe 5000 15
pause
```

在测试中，首先创建了一颗完全二叉树，如图 3 所示:

Initial Heap

value		32633	21892	16448	28007	16912	809	18509	20100	11815	7899	10908	1966	24996	27054	4985
address	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

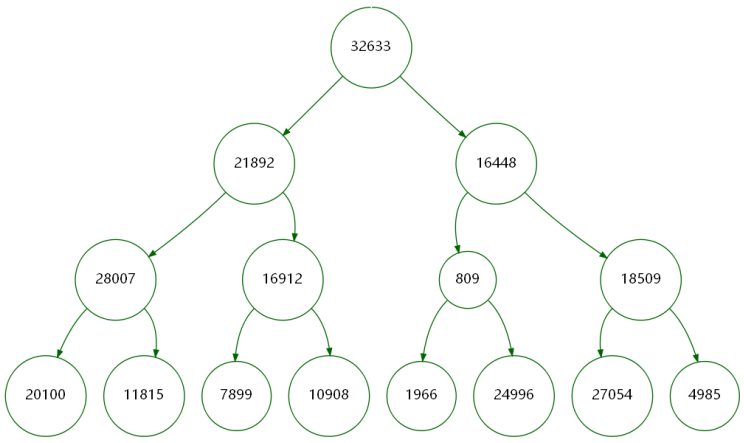

图 3 初始化堆的输出结果图

将图 3 中的完全二叉树调整为一个小顶堆，输出结果如图 4 所示：

Adjust Heap

value		809	7899	1966	11815	10908	16448	4985	20100	28007	16912	21892	32633	24996	27054	18509
address	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

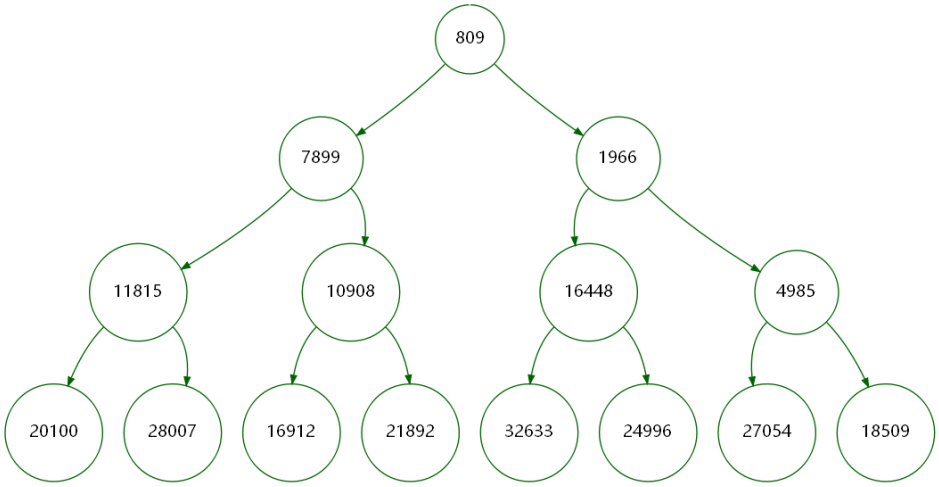

图 4 小顶堆的调整过程

最终得到的堆如图 5 所示，其中的元素即为 Top-K 问题的解(K=15)：

Final Heap

value		32750	32754	32753	32754	32759	32754	32753	32764	32763	32761	32760	32758	32764	32766	32755
address	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

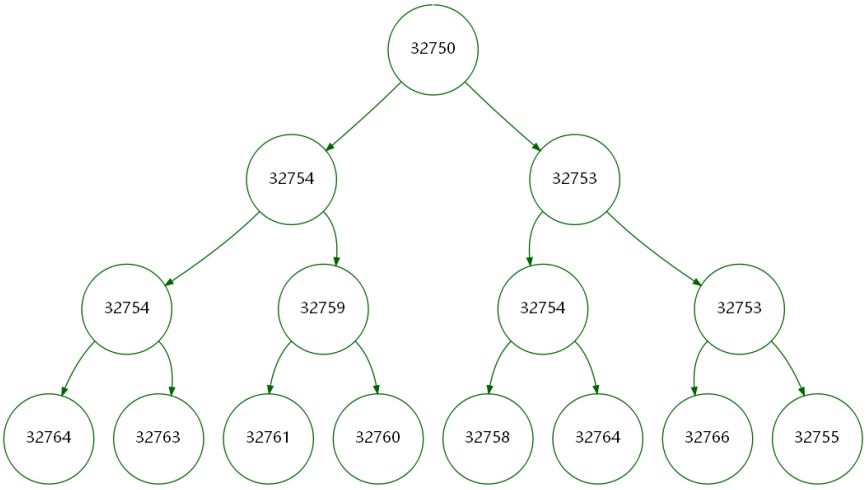

图 5 最终得到的堆(Top-K 问题的解)

八、总结及心得体会：

Top-K 问题是一类经典的问题，它能解决许多海量数据处理相关的问题。在本实验中，我们采用构造一个小顶堆来维护一个大小为 K 的数组，遍历整个数组的元素，将其与堆的根结点进行比较，如果遇到比根结点值更大的元素，则更新根结点值为该值，并将堆重新排序；如果遇到的值比根结点值小，则不更新根

结点的值。由于堆重排的时间复杂度为 O(logk)，因此整个算法的时间复杂度为 O(nlogk)。该解法的时间复杂度较低，并且这个解法的优点不仅可以找到第 K 大的元素，还能找出前 K 大的 K 个元素。

通过堆来解决 Top-K 问题已经是一个很好的方法了，既实现了快速查找又实现了快速移动元素。真要鸡蛋里挑骨头的话，Top-K 问题可能还有更好的解决办法：基于快速排序划分思想的二分法(BFPRT 算法)。

九、对本实验过程及方法、手段的改进建议：

1.本实验求解的是最大的 K 个数，可尝试求解最小的 K 个数(采用大顶堆实现)。

2.可尝试实现 Top-K 问题的其他解法(如上文中提到的 BFPRT 算法)。

3.实验过程中的堆调整过程可以更详细一点。

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。

五、实验器材（设备、元器件）

硬件平台

CPU：intel CORE i7-8750H

内存：DDR4 2666MHz 16G

GPU：NVIDIA GEFORCE GTX1070

硬盘：M.2 固态 512G + SSHD 1T

软件平台

操作系统：WIN10 1803

开发环境：C-Free 5, Graphviz 2.39

测试环境：Mingw5

六、实验步骤

具体步骤

①先取一点 v[0]作为起始点，初始化 dis[i],d[i]的值为 v[0]到其余点 v[i]的距离 w[0][i]，如果直接相邻初始化为权值，否则初始化为无限大；

②将 v[0]标记，vis[0] = 1(vis 一开始初始化为 0)；

③找寻与 v[0]相邻的最近点 v[k]，将 v[k]点记录下来，v[k]与 v[0]的距离记为 min；

④把 v[k]标记，vis[k]=1；

⑤查询并比较，让 dis[j]与 min+w[k][j]进行比较，判断是直接 v[0]连接 v[j]短，还是经过 v[k]连接 v[j]更短，即 dis[j]=MIN(dis[j],min+w[k][j])；

⑥继续重复步骤③与步骤⑤，知道找出所有点为止。

Dijkstra 算法主要代码：

```
void DijkstraPath(MGraph g,int *dist,int *path,int vs)   //vs 表示源顶点
{
    int i,j,k;
    bool *visited=(bool *)malloc(sizeof(bool)*g.n);
    for(i=0;i<g.n;i++)     //初始化
    {
        if(g.matrix[vs][i]>0&&i!=vs)
        {
            dist[i]=g.matrix[vs][i];
            path[i]=vs;     //path 记录最短路径上从 vs 到 i 的前一个顶点
        }
```

```
        else
        {
            dist[i]=INT_MAX;    //若 i 不与 vs 直接相邻，则权值置为无穷大
            path[i]=-1;
        }
        visited[i]=false;
        path[vs]=vs;
        dist[vs]=0;
    }
    FILE *fp=fopen("Dijkstra.gv","w+");
    fprintf(fp,"digraph Dijkstra {\nnode [shape=ellipse];\n");
    fprintf(fp,"v%d[shape=diamond,color=red,fontcolor=red];\n",vs);
    for (int i = 0; i < g.n && i != vs; i++)
        fprintf(fp,"v%d; ",i);
    for (int i = 0; i < g.n; i++)
    {
        for (int j = 0; j <g.n; j++)
        {
            if(g.matrix[i][j] != 0)
            {

    fprintf(fp,"v%d->v%d[style=bold,label=%d];\n",i,j,g.matrix[i][j]);
            }
        }
    }
    fprintf(fp,"}\n");
    fclose(fp);
    system("sfdp.exe -Tpng Dijkstra.gv -o DijkSetp01.png");
    visited[vs]=true;
    for(i=1; i<g.n; i++)     //循环扩展 n-1 次
    {
        int min=INT_MAX;
        int u;
        for(j=0;j<g.n;j++)    //寻找未被扩展的权值最小的顶点
        {
            if(visited[j]==false&&dist[j]<min)
            {
                min=dist[j];
                u=j;
            }
        }
        visited[u]=true;
        for(k=0;k<g.n;k++)   //更新 dist 数组的值和路径的值
        {

if(visited[k]==false&&g.matrix[u][k]>0&&min+g.matrix[u][k]<dist[k])
            {
                dist[k]=min+g.matrix[u][k];
                path[k]=u;
            }
        }
        Dijkstradot(g,path,visited,vs);
        char orderstr[128];
        sprintf(orderstr,"sfdp.exe       -Tpng       Dijkstra.gv       -o
DijkSetp%02d.png",i+1);
//      if( i == g.n-1)
            system(orderstr);
```

```
    }
}
```

七、实验数据及结果分析

使用.bat 脚本生成测试数据：

```
@echo off
call clean
@echo on
Dijkstra.exe 8 16
pause
```

初始化图，结果如图 1 所示：

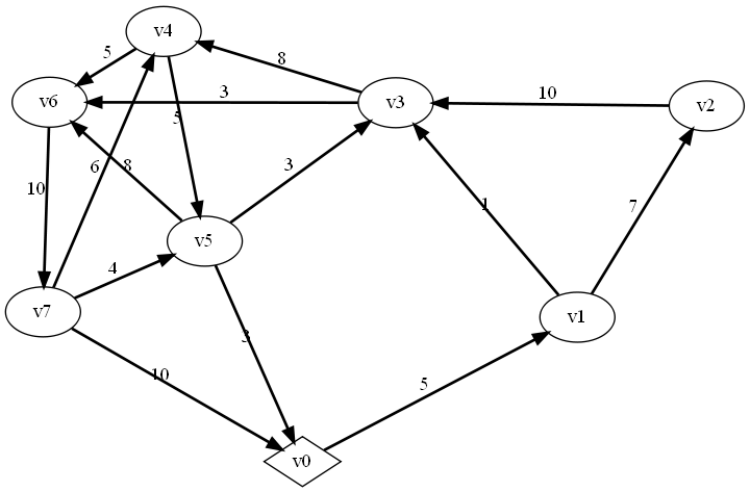

图 1 图的初始化的输出结果

选取v_0作为起始点，如图 2：

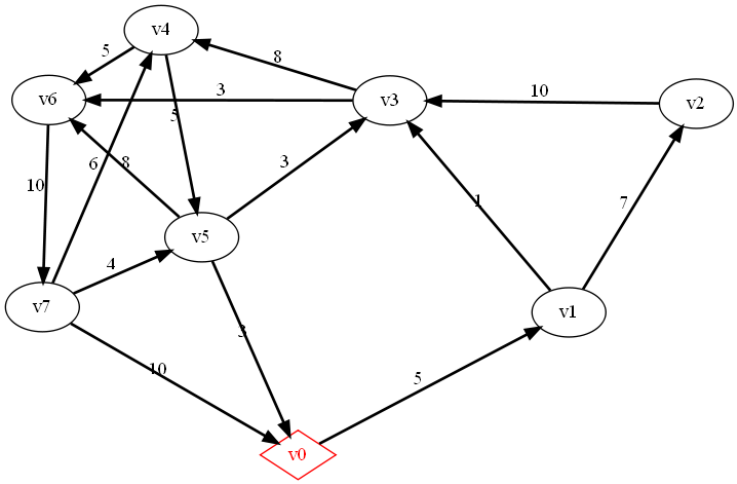

图 2 选取起始点

求v_0到其余各点的最短路径，如图 3 所示：

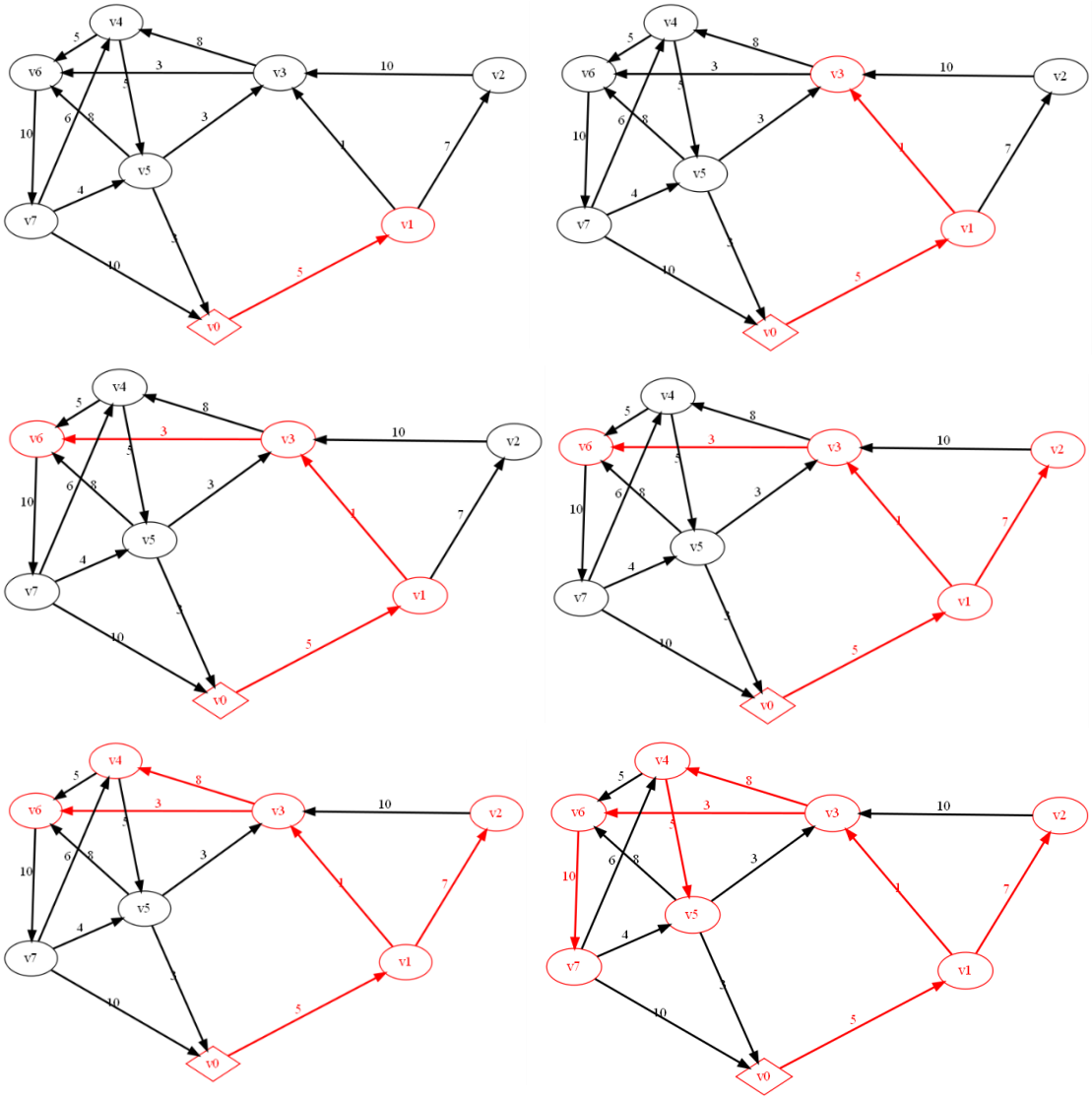

图 3 v_0到其余各点的最短路径示意图

八、总结及心得体会：

Dijkstra 算法的优点是时间复杂度相较于其他求最短路的算法来说更低，为$O(n^2)$，并且在使用堆优化(主要思想就是使用一个优先队列来代替最近距离的查找，用邻接表代替邻接矩阵)的情况下可以大幅节约时间开销，时间复杂度为$O(n\log_2 n)$。

Dijkstra 算法的缺点：在单源最短路径问题的某些实例中，可能存在权为负的边。如果$G=(V,E)$不包含从源s可达的负权回路，则对所有$v\in V$，最短路径的权定义$d(s,v)$仍然正确，即使它是一个负值也是如此。但如果存在一从s可达的负回路，最短路径的权的定义就不能成立。s到该回路上的结点就不存在最短路径。当有向图中出现负权时，则 Dijkstra 算法失效。

九、对本实验过程及方法、手段的改进建议：

1.对 Dijkstra 算法进行堆优化。

2.尝试解决当不存在源s可达的负回路这种情况下的最短路径求解。

3.可尝试扩展到全源最短路径求解。