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：刘文晨

学　　号：**2018080901006**

指导教师：周益民

日　　期：**2020 年 11 月 30 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。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->data 和树根关键字 root->data 进行比较，若 s->data =root->data,则无须插入，若 s->data<root->data,则插入到根的左子树中，若 s->data> root->data,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) 二叉排序树结点的删除

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 p->data 与二叉排序树的根节点的值 root->data 进行比较，若 p->data=root->data,则证明查找到待删除节点，进行下一步删除操作；若 p->data<root->data，则将待删除节点与根节点的左孩子的值进行比较；p->data<root->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) 二叉树节点插入

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) 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，这里有两种情况：

1. 若删除节点的值大于等于根的左孩子即的值 data>=root->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<root->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，这里有两种情况：

1. 若删除节点的值小于等于根的右孩子的值即 data<=root->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>root ->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™ i5-8300H CPU @ 2.30GHz　2.30GHz

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：Dev-C++ Version 5.11

编译器配置：TDM-GCC 4.9.2 64-bit Release

六、实验步骤

二叉排序树

```
1. **********二叉排序树**********
2. //二叉排序树结构定义
```

```
typedef struct node
{
    DataType data;
    struct node *lchild,*rchild;
    int flag;
}BSTNode,*BSTree;

********************相关操作********************
//初始化一个二叉排序树结点
BSTree InitNode(DataType data)
{
    BSTree Node=(BSTree)malloc(sizeof(BSTNode));
    Node->data=data;
    Node->lchild=NULL;
    Node->rchild=NULL;
    Node->flag=0;
    return Node;
}

//若在二叉排序树中不存在关键字等于 data 的元素，插入该元素
void InsertBST(BSTree *root, DataType data)
{
    if(*root==NULL)
    {
        *root=InitNode(data);
    }
    else
    {
        if(data<(*root)->data) InsertBST(&((*root)->lchild),data);
        else
        {
            if(data>(*root)->data) InsertBST(&((*root)->rchild),data);
        }
    }
}

//从文件输入元素的值，创建相应的二叉排序树
void CreateBST(BSTree *root,const char *filename)
{
    FILE *fp;
    DataType keynumber;
    *root=NULL;
```

```
45.     fp=fopen(filename,"r+");
46.     if(fp==NULL) exit(0x01);
47.     while(EOF!=fscanf(fp,"%d",&keynumber)) InsertBST(root,keynumb
    er);
48. }
49.
50. //删除排序二叉树, root 为指向二叉树根结点的指针
51. void DestroyBST(BSTree root)
52. {
53.     if(root!=NULL)
54.     {
55.         PreOrderCleanFlag(root->lchild);
56.         PreOrderCleanFlag(root->rchild);
57.         free(root);
58.     }
59. }
60.
61. //在根指针 root 所指二叉排序树 root 上，查找关键字等于 data 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
62. BSTree SearchBST(BSTree root,DataType data)
63. {
64.     BSTree q;
65.     q=root;
66.     while(q)
67.     {
68.         q->flag=1;
69.         if(q->data==data)
70.         {
71.             q->flag=2;
72.             return q;
73.         }
74.         if(q->data>data) q=q->lchild;
75.         else q=q->rchild;
76.     }
77.     return NULL;
78. }
79.
80. //在二叉排序树 root 中删去关键字为 value 的结点
81. BSTree DeleteBST(BSTree root,DataType value)
82. {
83.     BSTNode *p,*f,*s,*q;
84.     p=root;
85.     f=NULL;
86.     //查找关键字为 value 的待删结点 p
```

```
87.     while(p)
88.     {
89.         //找到则跳出循环,f 指向 p 结点的双亲结点
90.         if(p->data==value) break;
91.         f=p;
92.         if(p->data>value) p=p->lchild;
93.         else p=p->rchild;
94.     }
95.     //若找不到，返回原来的二叉排序树
96.     if(p==NULL)  return root;
97.     //若 p 无左子树
98.     if(p->lchild==NULL)
99.     {
100.            if(f==NULL) root=p->rchild;
101.            else
102.            {
103.                if(f->lchild==p) f->lchild=p->rchild;
104.                else f->rchild=p->rchild;
105.                //释放被删除的结点 p
106.                free(p);
107.            }
108.        }
109.        //若 p 有左子树
110.        else
111.        {
112.            q=p;
113.            s=p->lchild;
114.            while(s->rchild)
115.            {
116.                q=s;
117.                s=s->rchild;
118.            }
119.            if(q==p)
120.                q->lchild=s->lchild;
121.            else
122.                q->rchild=s->lchild;
123.            p->data=s->data;
124.            free(s);
125.        }
126.        return root;
127.    }
128.
129.    ********************四种遍历********************
130.    //先序遍历二叉树, root 为指向二叉树根结点的指针
```

```
131.    void PreOrderCleanFlag(BSTree root)
132.    {
133.        if(root!=NULL)
134.        {
135.            root->flag=0;
136.            PreOrderCleanFlag(root->lchild);
137.            PreOrderCleanFlag(root->rchild);
138.        }
139.    }
140.
141.    //中序遍历二叉树, root 为指向二叉树根结点的指针
142.    void InOrder(BSTree root)
143.    {
144.        if(root!=NULL)
145.        {
146.            InOrder(root->lchild);
147.            printf("%d ",root->data);
148.            InOrder(root->rchild);
149.        }
150.    }
151.
152.    //后序遍历二叉树, root 为指向二叉树根结点的指针
153.    void PostOrder(BSTree root)
154.    {
155.        if(root!=NULL)
156.        {
157.            PostOrder(root->lchild);
158.            PostOrder(root->rchild);
159.            printf("%d ",root->data);
160.        }
161.    }
162.
163.    //层序遍历, root 为指向二叉树根结点的指针
164.    void LevelOrder(BSTree root)
165.    {
166.        //定义一个队列
167.        BSTree Queue[MaxSize];
168.
169.        int front=-1,rear=0;
170.        // 若二叉树为空，遍历结束
171.        if(root==NULL) return;
172.        //根结点进入队列
173.        Queue[rear]=root;
174.        //若队列不为空，遍历，否则，遍历结束
```

```
175.        while(rear!=front)
176.        {
177.            //出队，打印出队结点的值
178.            front++;
179.            printf("%d ",Queue[front]->data);
180.            //若有左孩子，左孩子进入队列
181.            if(Queue[front]->lchild!=NULL)
182.            {
183.                rear++;
184.                Queue[rear]=Queue[front]->lchild;
185.            }
186.            //若有右孩子，右孩子进入队列
187.            if(Queue[front]->rchild!=NULL)
188.            {
189.                rear++;
190.                Queue[rear]=Queue[front]->rchild;
191.            }
192.        }
193.    }
194.
195.    ********************常见操作********************
196.    //在根指针 root 所指二叉排序树中交换左右子树
197.    void Exchange(BSTree root)
198.    {
199.        if(root==NULL) return;
200.        if(root->lchild==NULL && root->rchild==NULL) return;
201.        BSTree temp=root->lchild;
202.        root->lchild=root->rchild;
203.        root->rchild=temp;
204.        Exchange(root->lchild);
205.        Exchange(root->rchild);
206.    }
207.
208.    //在根指针 root 所指二叉排序树中求树的深度
209.    int Depth(BSTree root)
210.    {
211.        if(root==NULL) return 0;
212.        return 1+max(Depth(root->lchild),Depth(root->rchild));
213.    }
214.
215.    int max(int a,int b)
216.    {
217.        if (a>b) return a;
218.        return b;
```

```
219.    }
220.
221.    //在根指针 root 所指二叉排序树中计算总结点个数
222.    int CountBiNode(BSTree root)
223.    {
224.        if(root==NULL) return 0;
225.        int left=CountBiNode(root->lchild);
226.        int right=CountBiNode(root->rchild);
227.        return left+right+1;
228.    }
229.
230.    //根指针 root 所指二叉排序树中计算叶子结点个数
231.    int CountLeaf(BSTree root)
232.    {
233.        if(root==NULL) return 0;
234.        if(root->rchild==NULL && root->lchild==NULL) return 1;
235.        return (CountLeaf(root->lchild)+CountLeaf(root->rchild));
236.    }
237.
238.    ********************可视化部分********************
239.    void DotOrderList(BSTree root,FILE *fp)
240.    {
241.        if(root==NULL)
242.            return;
243.        char lpoint=root->lchild ? ' ' : ' ';
244.        char rpoint=root->rchild ? ' ' : ' ';
245.        if(root->flag==1)
246.        {
247.            fprintf(fp,"%d[label = \"<l>%c|<d>%d|<r>%c\",color=gr
    een];\n",root->data,lpoint,root->data,rpoint);
248.        }
249.        else if(root->flag==2)
250.        {
251.            fprintf(fp,"%d[label = \"<l>%c|<d>%d|<r>%c\",color=re
    d,fontcolor=red];\n",root->data,lpoint,root->data,rpoint);
252.        }
253.        else
254.            fprintf(fp,"%d[label = \"<l>%c|<d>%d|<r>%c\"];\n",roo
    t->data,lpoint,root->data,rpoint);
255.        DotOrderList(root->lchild,fp);
256.        DotOrderList(root->rchild,fp);
257.    }
258.
```

```
259.    void DotOrderLink(BSTree root,FILE *fp)
260.    {
261.        if(root==NULL)
262.            return;
263.
264.        if(root->lchild)
265.            fprintf(fp,"%d:l:sw -> %d:d;\n",root->data,root->lchi
    ld->data);
266.
267.        if(root->rchild)
268.            fprintf(fp,"%d:r:se -> %d:d;\n",root->data,root->rchi
    ld->data);
269.
270.        DotOrderLink(root->lchild,fp);
271.        DotOrderLink(root->rchild,fp);
272.    }
273.
274.    void MakeDot(BSTree root,char *tital=NULL)
275.    {
276.        FILE *fp=fopen("bstree.gv","w+");
277.        fprintf(fp,"digraph BSTree {\n");
278.        if(tital != NULL)
279.        {
280.            fprintf(fp,"labelloc = t; labeljust = l;\n");
281.            fprintf(fp,"label = \"%s\";\n",tital);
282.        }
283.        fprintf(fp,"node [fontname = Verdana, color=navy, shape=r
    ecord, height=.1];\n");
284.        fprintf(fp,"edge [fontname = Verdana, color=navy, style=s
    olid];\n");
285.        DotOrderList(root,fp);
286.        DotOrderLink(root,fp);
287.        fprintf(fp,"}\n\n");
288.        fclose(fp);
289.    }
```

平衡二叉树

```
1. ********************平衡二叉树********************
2. //平衡二叉树结构定义
3. typedef struct node
4. {
5.     DataType data;
6.     int bf;
```

```
    int flag;
    struct node *lchild,*rchild;
}AVLNode,*AVLTree;

********************相关操作********************
//初始化一个平衡二叉树结点
AVLTree InitNode(DataType data)
{
    AVLTree Node=(AVLTree)malloc(sizeof(AVLNode));
    Node->data=data;
    Node->lchild=NULL;
    Node->rchild=NULL;
    Node->bf=0;
    Node->flag=0;
}

//在平衡二叉树中插入值为 data 的元素，使之成为一棵新的平衡二叉排序树
void InsertAVL(AVLTree *root,DataType data)
{
    AVLNode *s;
    AVLNode *a,*fa,*p,*fp,*b,*c;
    s=InitNode(data);
    if(*root==NULL) *root=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=*root;
        fa=NULL;
        p=*root;
        fp=NULL;
        while(p!=NULL)
        {
            if(p->bf!=0)
            {
                a=p;
                fa=fp;
            }
            fp=p;
            if (data<p->data) p=p->lchild;
            else if(data>p->data) p=p->rchild;
            else
            {
                free(s);
```

```
                return;
            }
        }
        //插入 S
        if(data<fp->data) fp->lchild=s;
        else fp->rchild=s;
        //确定结点 B，并修改 A 的平衡因子
        if (data<a->data)
        {
            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(data<p->data)
            {
                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) *root=b;
            else
            {
                if(a==fa->lchild) fa->lchild=b;
                else fa->rchild=b;
            }
```

```
94.         }
95.         else if(a->bf==2 && b->bf==-1)        /* LR 型 */
96.         {
97.             b=a->lchild;
98.             c=b->rchild;
99.             b->rchild=c->lchild;
100.                a->lchild=c->rchild;
101.                c->lchild=b;
102.                c->rchild=a;
103.                if(s->data<c->data)
104.                {
105.                    a->bf=-1;
106.                    b->bf=0;
107.                    c->bf=0;
108.                }
109.                else if(s->data>c->data)
110.                {
111.                    a->bf=0;
112.                    b->bf=1;
113.                    c->bf=0;
114.                }
115.                else
116.                {
117.                    a->bf=0;
118.                    b->bf=0;
119.                }
120.                if(fa==NULL) *root=c;
121.                else if(a==fa->lchild) fa->lchild=c;
122.                else fa->rchild=c;
123.                }
124.            else if(a->bf==-2 && b->bf==1)        /* RL 型 */
125.            {
126.                b=a->rchild;
127.                c=b->lchild;
128.                b->lchild=c->rchild;
129.                a->rchild=c->lchild;
130.                c->lchild=a;
131.                c->rchild=b;
132.                if(s->data<c->data)
133.                {
134.                    a->bf=0;
135.                    b->bf=-1;
136.                    c->bf=0;
137.                }
```

```
                else if(s->data>c->data)
                {
                    a->bf=1;
                    b->bf=0;
                    c->bf=0;
                }
                else
                {
                    a->bf=0;
                    b->bf=0;
                }
                if (fa==NULL) *root=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) *root=b;
                else if(a==fa->lchild) fa->lchild=b;
                else fa->rchild=b;
            }
        }
    }

    //在平衡二叉树 root 中删去关键字为 value 的结点
    AVLTree DeleteAVL(AVLTree root,DataType value)
    {
        AVLNode *p,*f,*s,*q;
        p=root;
        f=NULL;
        //查找关键字为 value 的待删结点 p
        while(p)
        {
            //找到则跳出循环,f 指向 p 结点的双亲结点
            if(p->data==value) break;
            f=p;
            if(p->data>value) p=p->lchild;
            else p=p->rchild;
        }
```

```
182.        //若找不到，返回原来的平衡二叉树
183.        if(p==NULL)  return root;
184.        //若 p 无左子树
185.        if(p->lchild==NULL)
186.        {
187.            if(f==NULL) root=p->rchild;
188.            else
189.            {
190.                if(f->lchild==p) f->lchild=p->rchild;
191.                else f->rchild=p->rchild;
192.                //释放被删除的结点 p
193.                free(p);
194.            }
195.        }
196.        //若 p 有左子树
197.        else
198.        {
199.            q=p;
200.            s=p->lchild;
201.            while(s->rchild)
202.            {
203.                q=s;
204.                s=s->rchild;
205.            }
206.            if(q==p)
207.                q->lchild=s->lchild;
208.            else
209.                q->rchild=s->lchild;
210.            p->data=s->data;
211.            free(s);
212.        }
213.        return root;
214.    }
215.
216.    //从文件输入元素的值，创建相应的平衡二叉树
217.    void CreateAVL(AVLTree *root,const char *filename)
218.    {
219.        FILE *fp;
220.        DataType keynumber;
221.        *root=NULL;
222.        fp=fopen(filename,"r+");
223.        while(EOF!=fscanf(fp,"%d",&keynumber)) InsertAVL(root,key
    number);
224.    }
```

```
225.
226.    //删除平衡二叉树, root 为指向二叉树根结点的指针
227.    void DestroyAVL(AVLTree root)
228.    {
229.        if(root!=NULL)
230.        {
231.            PreOrderCleanFlag(root->lchild);
232.            PreOrderCleanFlag(root->rchild);
233.            free(root);
234.        }
235.    }
236.
237.    //在根指针 root 所指平衡二叉树 root 上，查找关键字等于 data 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
238.    AVLTree SearchAVL(AVLTree root,DataType data)
239.    {
240.        AVLTree q;
241.        q=root;
242.        while(q)
243.        {
244.            q->flag=1;
245.            if(q->data==data)
246.            {
247.                q->flag=2;
248.                return q;
249.            }
250.            if(q->data>data) q=q->lchild;
251.            else q=q->rchild;
252.        }
253.        return NULL;
254.    }
255.
256.    ********************四种遍历********************
257.    //先序遍历二叉树, root 为指向二叉树根结点的指针
258.    void PreOrderCleanFlag(AVLTree root)
259.    {
260.        if(root!=NULL)
261.        {
262.            //printf("%d(%d)\t",root->data,root->bf);
263.            root->flag=0;
264.            PreOrderCleanFlag(root->lchild);
265.            PreOrderCleanFlag(root->rchild);
266.        }
267.    }
```

```
268.
269.    //中序遍历二叉树，root 为指向二叉树根结点的指针
270.    void InOrder(AVLTree root)
271.    {
272.        if(root!=NULL)
273.        {
274.            InOrder(root->lchild);
275.            printf("%d ",root->data);
276.            InOrder(root->rchild);
277.        }
278.    }
279.
280.    //后序遍历二叉树，root 为指向二叉树根结点的指针
281.    void PostOrder(AVLTree root)
282.    {
283.        if(root!=NULL)
284.        {
285.            PostOrder(root->lchild);
286.            PostOrder(root->rchild);
287.            printf("%d ",root->data);
288.        }
289.    }
290.
291.    //层序遍历，root 为指向二叉树根结点的指针
292.    void LevelOrder(AVLTree root)
293.    {
294.        //定义一个队列
295.        AVLTree Queue[MaxSize];
296.
297.        int front=-1,rear=0;
298.        // 若二叉树为空，遍历结束
299.        if(root==NULL) return;
300.        //根结点进入队列
301.        Queue[rear]=root;
302.        //若队列不为空，遍历，否则，遍历结束
303.        while(rear!=front)
304.        {
305.            //出队，打印出队结点的值
306.            front++;
307.            printf("%d ",Queue[front]->data);
308.            //若有左孩子，左孩子进入队列
309.            if(Queue[front]->lchild!=NULL)
310.            {
311.                rear++;
```

```
312.                 Queue[rear]=Queue[front]->lchild;
313.             }
314.             //若有右孩子，右孩子进入队列
315.             if(Queue[front]->rchild!=NULL)
316.             {
317.                 rear++;
318.                 Queue[rear]=Queue[front]->rchild;
319.             }
320.         }
321.     }
322.
323.     ********************常见操作********************
324.     //在根指针 root 所指平衡二叉树中交换左右子树
325.     void Exchange(AVLTree root)
326.     {
327.         if(root==NULL) return;
328.         if(root->lchild==NULL && root->rchild==NULL) return;
329.         AVLTree temp=root->lchild;
330.         root->lchild=root->rchild;
331.         root->rchild=temp;
332.         Exchange(root->lchild);
333.         Exchange(root->rchild);
334.     }
335.
336.     //在根指针 root 所指平衡二叉树中求树的深度
337.     int Depth(AVLTree root)
338.     {
339.         if(root==NULL) return 0;
340.         return 1+max(Depth(root->lchild),Depth(root->rchild));
341.     }
342.
343.     int max(int a,int b)
344.     {
345.         if (a>b) return a;
346.         return b;
347.     }
348.
349.     //在根指针 root 所指平衡二叉树中计算总结点个数
350.     int CountBiNode(AVLTree root)
351.     {
352.         if(root==NULL) return 0;
353.         int left=CountBiNode(root->lchild);
354.         int right=CountBiNode(root->rchild);
355.         return left+right+1;
```

```
356.    }
357.
358.    //根指针 root 所指平衡二叉树中计算叶子结点个数
359.    int CountLeaf(AVLTree root)
360.    {
361.        if(root==NULL) return 0;
362.        if(root->rchild==NULL && root->lchild==NULL) return 1;
363.        return (CountLeaf(root->lchild)+CountLeaf(root->rchild));

364.    }
365.
366.    ********************可视化部分********************
367.    void DotOrderList(AVLTree root,FILE *fp)
368.    {
369.        if(root==NULL)
370.            return;
371.        char lpoint=root->lchild ? ' ' : ' ';
372.        char rpoint=root->rchild ? ' ' : ' ';
373.        if(root->flag==1)
374.        {
375.            fprintf(fp,"%d[label = \"<l>%c|<d>%d|<r>%c\",color=gr
    een];\n",root->data,lpoint,root->data,rpoint);
376.        }
377.        else if(root->flag==2)
378.        {
379.            fprintf(fp,"%d[label = \"<l>%c|<d>%d|<r>%c\",color=re
    d,fontcolor=red];\n",root->data,lpoint,root->data,rpoint);
380.        }
381.        else
382.            fprintf(fp,"%d[label = \"<l>%c|<d>%d|<r>%c\"];\n",roo
    t->data,lpoint,root->data,rpoint);
383.        DotOrderList(root->lchild,fp);
384.        DotOrderList(root->rchild,fp);
385.    }
386.
387.    void DotOrderLink(AVLTree root,FILE *fp)
388.    {
389.        if(root==NULL)
390.            return;
391.
392.        if(root->lchild)
393.            fprintf(fp,"%d:l:sw -> %d:d;\n",root->data,root->lchi
    ld->data);
394.
```

```
395.         if(root->rchild)
396.             fprintf(fp,"%d:r:se -> %d:d;\n",root->data,root->rchi
   ld->data);
397.
398.         DotOrderLink(root->lchild,fp);
399.         DotOrderLink(root->rchild,fp);
400.     }
401.
402.     void MakeDot(AVLTree root,char *tital=NULL)
403.     {
404.         FILE *fp=fopen("avltree.gv","w+");
405.         fprintf(fp,"digraph BSTree {\n");
406.         if(tital != NULL)
407.         {
408.             fprintf(fp,"labelloc = t; labeljust = l;\n");
409.             fprintf(fp,"label = \"%s\";\n",tital);
410.         }
411.         fprintf(fp,"node [fontname = Verdana, color=navy, shape=r
   ecord, height=.1];\n");
412.         fprintf(fp,"edge [fontname = Verdana, color=navy, style=s
   olid];\n");
413.         DotOrderList(root,fp);
414.         DotOrderLink(root,fp);
415.         fprintf(fp,"}\n\n");
416.         fclose(fp);
417. }
```

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。

822 737 408 24 897 340 36 318 552 824

863 482 471 214 301 363 641 62 34 649

854 940 96 281 338 37 71 28 302 516

898 674 757 735 174 721 160 486 370 990

764 986 539 552 143 165 285 632 827 695

303 319 456 48 376 128 398 911 579 853

686 164 909 337 390 165 609 553 365 792

699 116 342 803 564 731 12 748 125 138

779 709 911 540 219 924 990 497 610 277

826 259 812 465 634 760 836 409 223 340

得到的排序二叉树如图 1 所示。

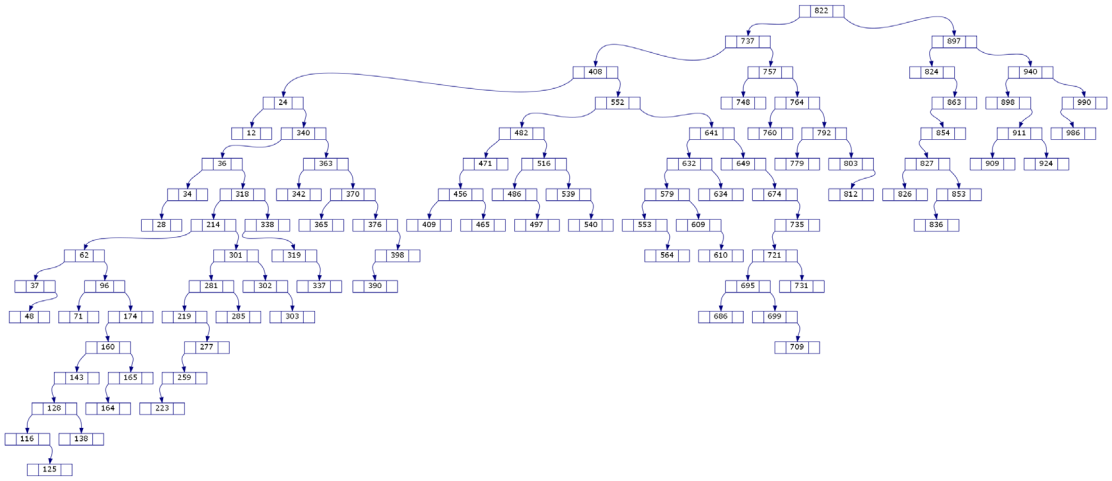


图 1 二叉排序树生成结果图

该排序二叉树的深度、总结点数、叶子结点数以及中序遍历的结果如图 2 所示。

```
C:\Users\81228\Desktop\BSTreeByXiye>BSTree.exe data.txt
该二叉排序树的深度为： 16
该二叉排序树的总结点数为： 95
该二叉排序树的叶子结点数为： 33
中序遍历的结果为:
12 24 28 34 36 37 48 62 71 96 116 125 128 138 143 160 164 165 174 214 219 223 259 277 281 285 301 302
303 318 319 337 338 340 342 363 365 370 376 390 398 408 409 456 465 471 482 486 497 516 539 540 552 55
3 564 579 609 610 632 634 641 649 674 686 695 699 709 721 731 735 737 748 757 760 764 779 792 803 812
822 824 826 827 836 853 854 863 897 898 909 911 924 940 986 990
C:\Users\81228\Desktop\BSTreeByXiye>REM dot.exe -Tpng bstree.gv -o dotbst.png && dotbst.png

C:\Users\81228\Desktop\BSTreeByXiye>REM fdp -Tpng bstree.gv -o fdpbst.png && fdpbst.png

C:\Users\81228\Desktop\BSTreeByXiye>REM neato -Tpng bstree.gv -o neatobst.png && neatobst.png

C:\Users\81228\Desktop\BSTreeByXiye>REM sfdp -Tpng bstree.gv -o sfdpbst.png sfdpbst.png

C:\Users\81228\Desktop\BSTreeByXiye>pause
请按任意键继续. . .
```

图 2 二叉排序树的深度、总结点数、叶子节点数以及中序遍历结果图

在该排序二叉树中查找值 62 的结点，查命中。搜索过程如图 3 所示。

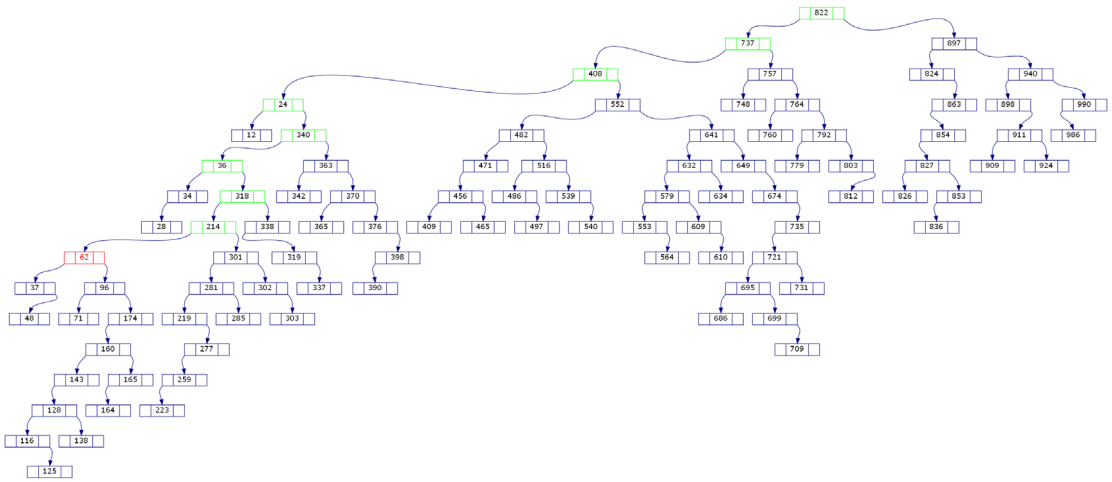


图 3 查找值为 62 的结点搜索过程图

在该排序二叉树中查找值 98 的结点，查不命中。搜索过程如图 4 所示。

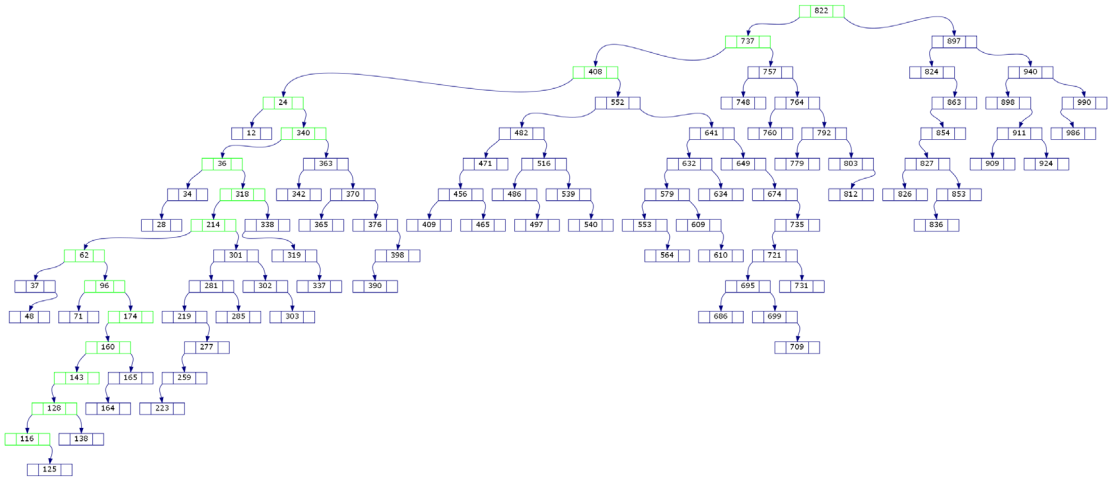


图 4 查找值为 98 的结点搜索过程图

对于二叉排序树的删除，如果删除结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结点，选择对根结点即 data 为 822 的结点进行删除。为了维持二叉排序树的排序性，算法会将待删除结点的左子树的最大值替代该结点的值，并且将待删除结点的左子树最大值所在的结点进行删除。删除前的二叉排序树如图 1 所示，删除后的二叉排序树如图 5 所示。

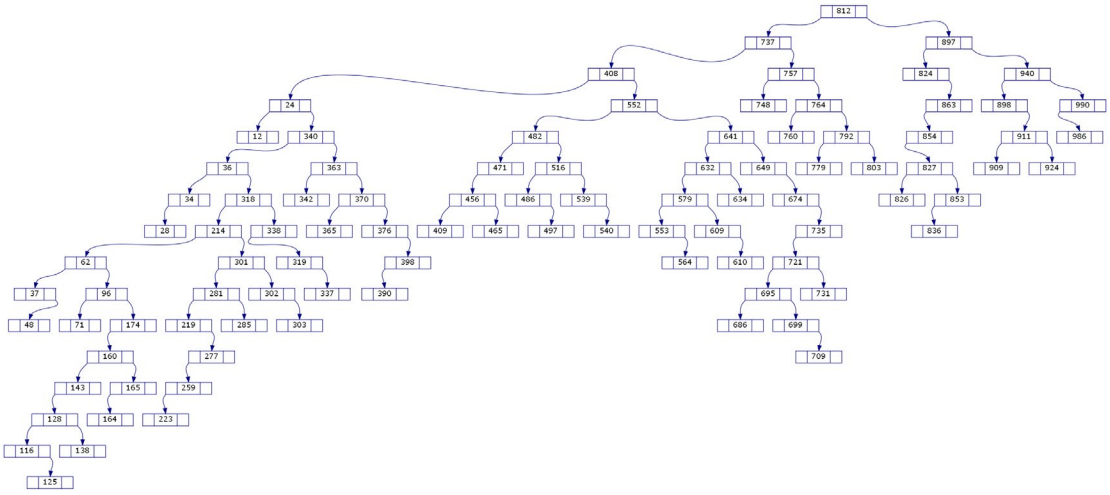


图 5 二叉排序树删除结点示意图

对于同样的输入序列，生成得到的平衡二叉树如图 6 所示。

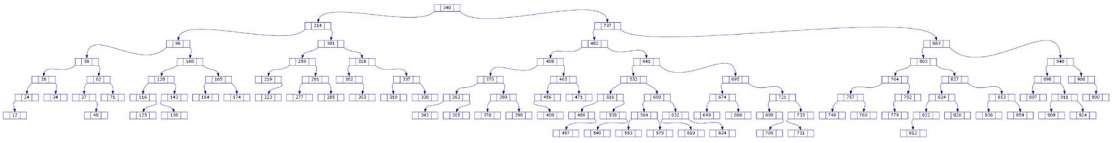


图 6 平衡二叉树生成结果图

对比图 1 和图 6，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。

该平衡二叉树的深度、总结点数、叶子结点数以及中序遍历的结果如图 7 所示。

```
C:\Users\81228\Desktop\AVLTreeByXiye>AVLTree.exe data.txt
该平衡二叉树的深度为： 8
该平衡二叉树的总结点数为： 95
该平衡二叉树的叶子结点数为： 41
中序遍历的结果为：
12 24 28 34 36 37 48 62 71 96 116 125 128 138 143 160 164 165 174 214 219 223 259 277 281 285 301 302 303 318 319 337 33
8 340 342 363 365 370 376 390 398 408 409 456 465 471 482 486 497 516 539 540 552 553 564 579 609 610 632 634 641 649 67
4 686 695 699 709 721 731 735 737 748 757 760 764 779 792 803 812 822 824 826 827 836 853 854 863 897 898 909 911 924 94
0 986 990
C:\Users\81228\Desktop\AVLTreeByXiye>REM dot.exe -Tpng avltree.gv -o dotbst.png && dotbst.png

C:\Users\81228\Desktop\AVLTreeByXiye>REM fdp -Tpng avltree.gv -o fdpbst.png && fdpbst.png

C:\Users\81228\Desktop\AVLTreeByXiye>REM neato -Tpng avltree.gv -o neatobst.png && neatobst.png

C:\Users\81228\Desktop\AVLTreeByXiye>REM sfdp -Tpng avltree.gv -o sfdpbst.png sfdpbst.png

C:\Users\81228\Desktop\AVLTreeByXiye>pause
请按任意键继续. . .
```

图 7 平衡二叉树的深度、总结点数、叶子节点数以及中序遍历结果图

在该排序二叉树中查找值 709 的结点，查命中。搜索过程如图 8 所示。

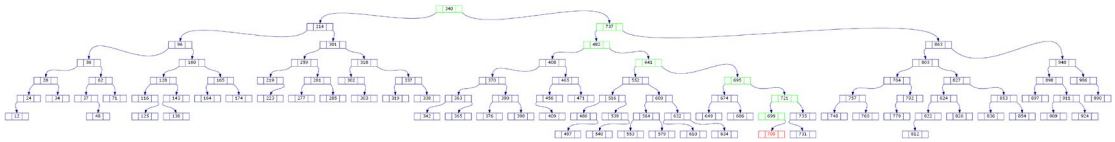


图 8 查找值为 709 的结点搜索过程图

在该排序二叉树中查找值 98 的结点，查不命中。搜索过程如图 9 所示。

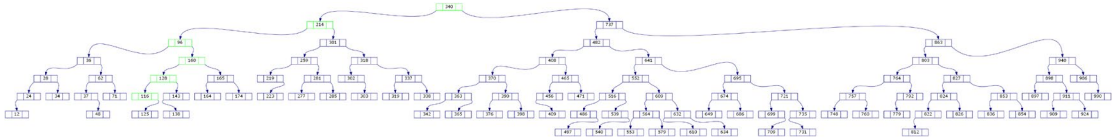


图 9 查找值为 98 的结点搜索过程图

对于平衡二叉树的删除，选择对根结点即 data 为 340 的结点进行删除。为了维持平衡二叉树的排序性，算法会将待删除结点的左子树的最大值替代该结点的值，并且将待删除结点的左子树最大值所在的结点进行删除。删除前的平衡二叉树如图 6 所示，删除后的平衡二叉树如图 10 所示。

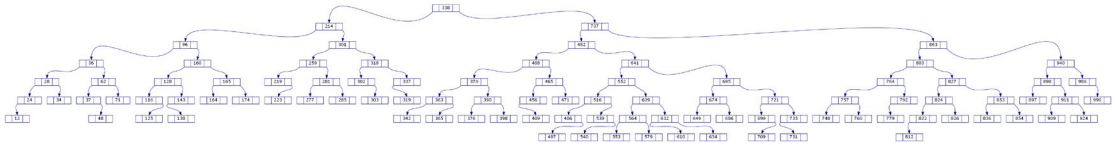


图 10 平衡二叉树删除结点示意图

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、求树的深度、总结点数、叶子结点数等功能。

在实现二叉排序树的修改操作时，我没有意识到二叉排序树不能直接修改的结点的值，这样做会破坏二叉排序树的排序性。我认为，正确的方法是删除 oldValue 结点并插入 newValue 结点。

在实现平衡二叉树的删除操作时，我简单地认为这与二叉排序树的删除操作基本一致。后来在真正实现时，我发现先删除结点再维护平衡性的难度巨大，于是改为递归查找并删除，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：

1. 加强 dot 语言的学习，改进可视化部分的代码，使生成的图片更加美观；
2. 将平衡二叉树的平衡旋转部分的代码封装成一个新的函数，提高代码的

可读性；

3. 更加完善代码注释，提高代码的可读性。

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）

处理器：Intel® Core™ i5-8300H CPU @ 2.30GHz　2.30GHz

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：Dev-C++ Version 5.11

编译器配置：TDM-GCC 4.9.2 64-bit Release

六、实验步骤

Top-K 算法

```
1. **********Top-K 算法**********
2. //交换函数
3. void swap(int *a,int *b)
4. {
```

```
5.      *a=*a+*b;
6.      *b=*a-*b;
7.      *a=*a-*b;
8.  }
9.
10. //堆遍历
11. void HeapTraverse(int a[],int size)
12. {
13.      for(int i=0;i<size;i++) printf("%d ",a[i]);
14.      printf("\n");
15. }
16.
17. //带起始点的堆调整
18. void HeapShift(int a[],int size,int start)
19. {
20.      int dad=start,son=2*dad+1;
21.
22.      while(son<=size)
23.      {
24.          // 找子结点的最大值
25.          if(son+1<=size && a[son]>a[son+1]) son++;
26.
27.          if(a[dad]<a[son]) return;
28.          else
29.          {
30.              swap(&a[dad],&a[son]);
31.              dad=son;
32.              son=2*dad+1;
33.          }
34.      }
35. }
36.
37. //堆排序
38. void HeapSort(int a[],int size)
39. {
40.      //构建初始堆
41.      for(int i=size/2-1;i>=0;i--)
42.      {
43.          HeapShift(a,size-1,i);
44.      }
45.      for(int j=size-1;j>0;j--)
46.      {
47.          // 堆顶元素和堆中的最后一个元素交换
48.          swap(&a[0],&a[j]);
```

```
49.          // 重新调整结构，使其继续满足堆定义
50.          HeapShift(a,j-1,0);
51.      }
52. }
53.
54. //堆调整
55. void HeapAdjust(int a[],int i,int size)
56. {
57.      int child;
58.      int temp;
59.      for(;2*i+1<size;i=child)
60.      {
61.          child=2*i+1;
62.          if(child<size-1 && a[child+1]<a[child]) child++;
63.          if(a[i]>a[child])
64.          {
65.              temp=a[i];
66.              a[i]=a[child];
67.              a[child]=temp;
68.          }
69.          else break;
70.      }
71. }
72.
73. ********************可视化部分********************
74. void DotHeap(int heap[],int n,char *label,int input=-1,int drop=-
    1)
75. {
76.      FILE *fpTree=fopen("heapT.gv","w+");
77.      fprintf(fpTree,"digraph heapT {\n");
78.      fprintf(fpTree,"fontname = \"Microsoft YaHei\"; labelloc = t;
     labeljust = l; rankdir = TB;\n");
79.      fprintf(fpTree,"label = \"%s\";\n",label);
80.      fprintf(fpTree,"node [fontname = \"Microsoft YaHei\", color=d
    arkgreen, shape=circle, height=.1];\n");
81.      fprintf(fpTree,"edge [fontname = \"Microsoft YaHei\", color=d
    arkgreen, style=solid, arrowsize=0.7];\n");
82.
83.      if(input!=-1 && drop!=-1)
84.      {
85.          fprintf(fpTree,"in%d[label=\"%d\",shape=Mcircle,fontcolor
    =blue,color=blue];\n",input,input);
86.      }
87.
```

```
88.     for(int i=0;i<n;i++)
89.     {
90.         fprintf(fpTree,"%d[label=\"%d\"];\n",heap[i],heap[i]);
91.     }
92.
93.     if(input!=-1 && drop!=-1 && input!=drop)
94.     {
95.         fprintf(fpTree,"%d[label=\"%d\",shape=circle,fontcolor=bl
    ue,color=blue];\n",input,input);
96.     }
97.
98.     if(input!=-1 && drop!=-1)
99.     {
100.            if(input==drop)
101.            {
102.                fprintf(fpTree,"%d[label=\"%d\",shape=doublecircl
    e,fontcolor=darkgreen,color=darkgreen];\n",heap[0],heap[0]);
103.            }
104.        }
105.
106.        if(input!=-1 && drop!=-1)
107.        {
108.            if(input!=drop)
109.            {
110.                fprintf(fpTree,"dp%d[label=\"%d\",shape=doublecir
    cle,fontcolor=red,color=red];\n",drop,drop);
111.            }
112.            else
113.            {
114.                fprintf(fpTree,"dp%d[label=\"%d\",shape=Mcircle,f
    ontcolor=red,color=red];\n",drop,drop);
115.            }
116.        }
117.
118.        if(input!=-1 && drop!=-1)
119.        {
120.            fprintf(fpTree,"{rank = same; in%d; %d; dp%d;};\n",in
    put,heap[0],drop);
121.            fprintf(fpTree,"in%d -> %d[color=blue];\n",input,heap
    [0]);
122.            fprintf(fpTree,"%d -> dp%d[color=red];\n",heap[0],dro
    p);
123.        }
124.
```

```
125.         for(int i=0;i<n;i++)
126.         {
127.             if(2*(i+1)-1<n) fprintf(fpTree,"%d:sw -> %d;\n",heap[
     i],heap[2*(i+1)-1]);
128.             if(2*(i+1)<n) fprintf(fpTree,"%d:se -> %d;\n",heap[i]
     ,heap[2*(i+1)]);
129.         }
130.
131.         fprintf(fpTree,"node [fontname = \"Microsoft YaHei\", col
     or=darkgreen, shape=record, height=.1];\n");
132.         fprintf(fpTree,"edge [fontname = \"Microsoft YaHei\", col
     or=darkgreen, style=solid];\n");
133.         fprintf(fpTree,"struct [ label = \"{value|address} |");
134.         fprintf(fpTree,"{|%d} ",0);
135.         for(int i=0;i<n;i++)
136.         {
137.             fprintf(fpTree,"| {%d|%d} ",heap[i],i+1);
138.         }
139.         fprintf(fpTree,"\"]; \n");
140.         fprintf(fpTree,"%d -> struct[color=white]; \n",heap[n-1])
     ;
141.         fprintf(fpTree,"}\n\n");
142.         fclose(fpTree);
143. }
```

七、实验数据及结果分析

选取 k=10 的一列，以 N 作为横坐标，t 为纵坐标，绘制散点图 1 所示。

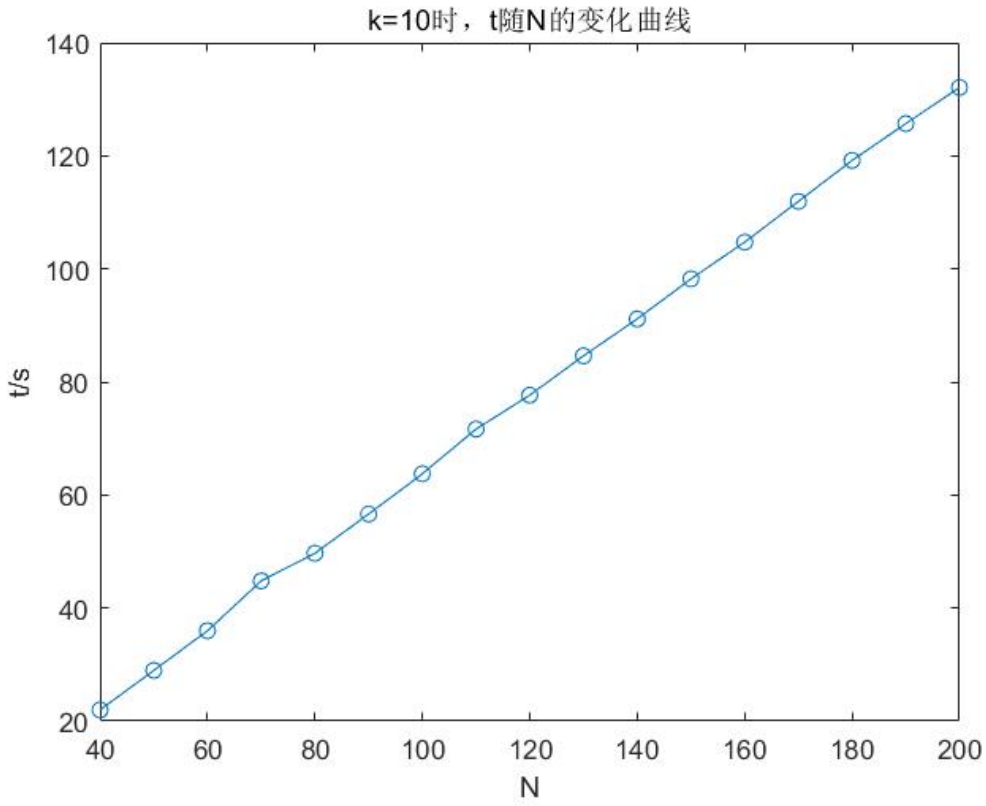


图 1 k=10 时，t 随 N 的变化曲线

如图 1 所示，在 k 值确定时，时间 t 与 N 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。

确定 N=1000 时，以 log2（k）作为横坐标，t 为纵坐标，绘制散点图如图 2 所示，在 N 值确定的时候，时间 t 与 k 的对数约成线性关系。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。

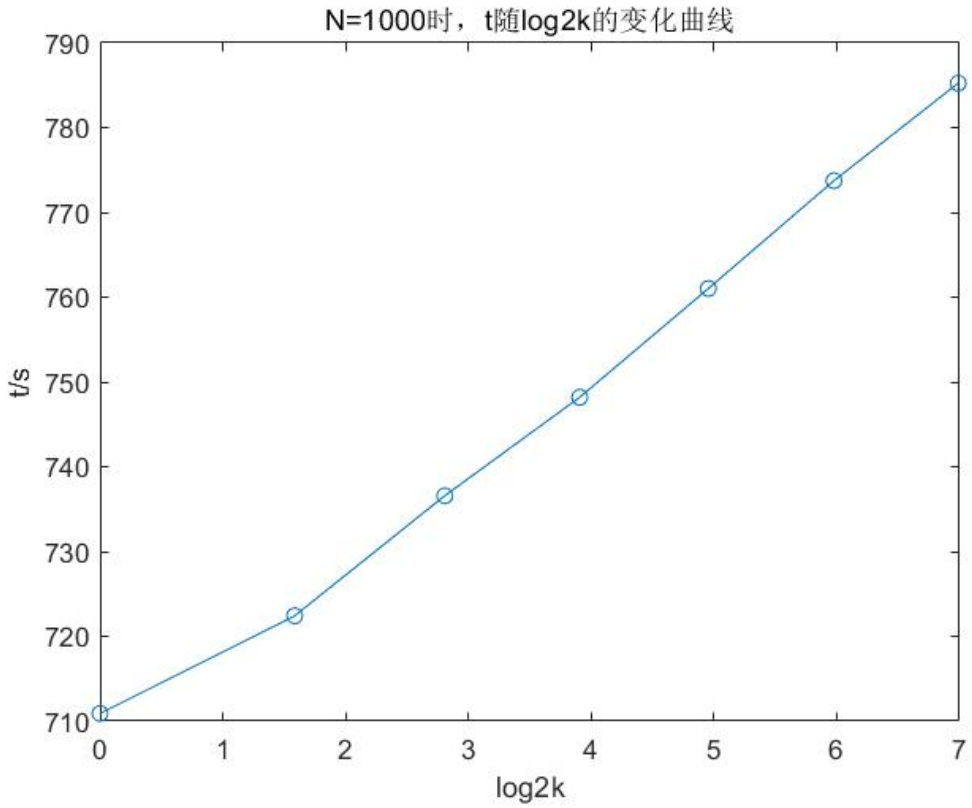
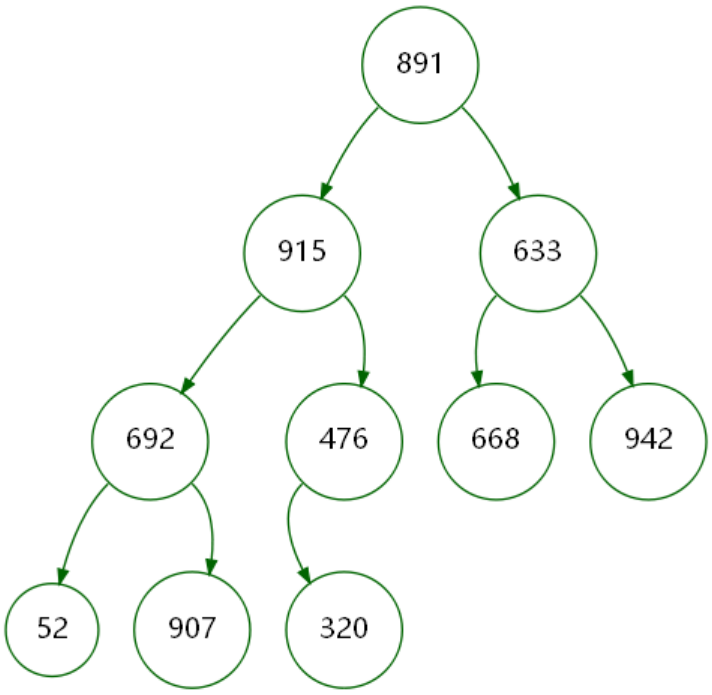


图 2 N=1000 时，t 随 log2k 的变化曲线

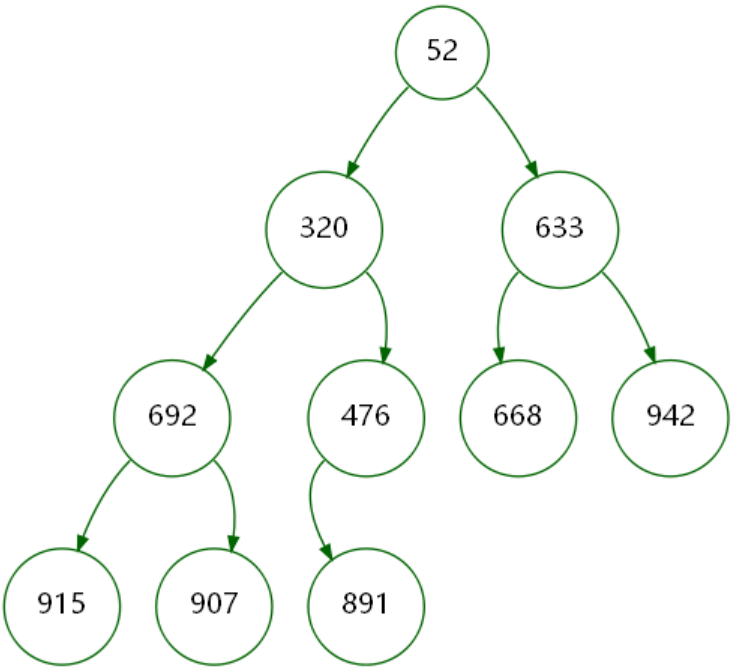
在测试中，构造了堆调整的每个过程，如图 3 所示。

Initial Heap



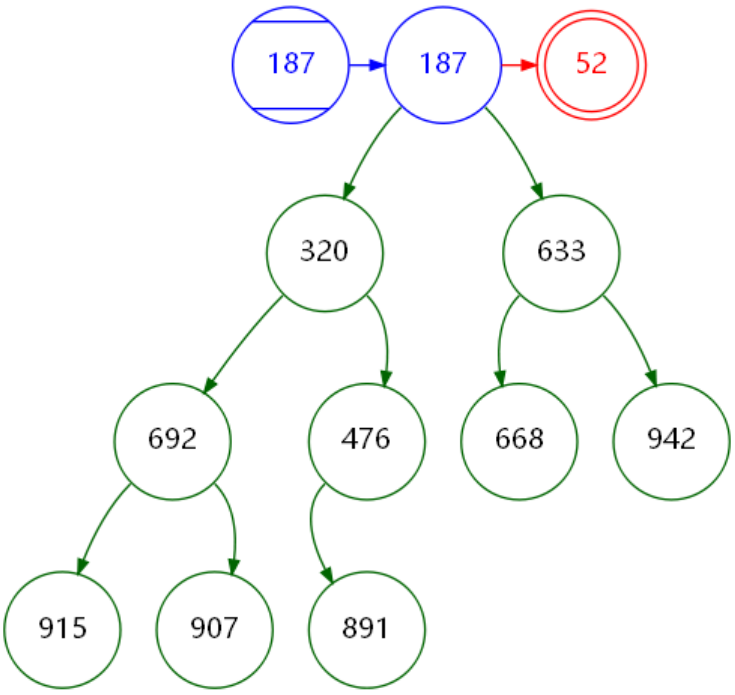
value		891	915	633	692	476	668	942	52	907	320
address	0	1	2	3	4	5	6	7	8	9	10

Adjust Heap



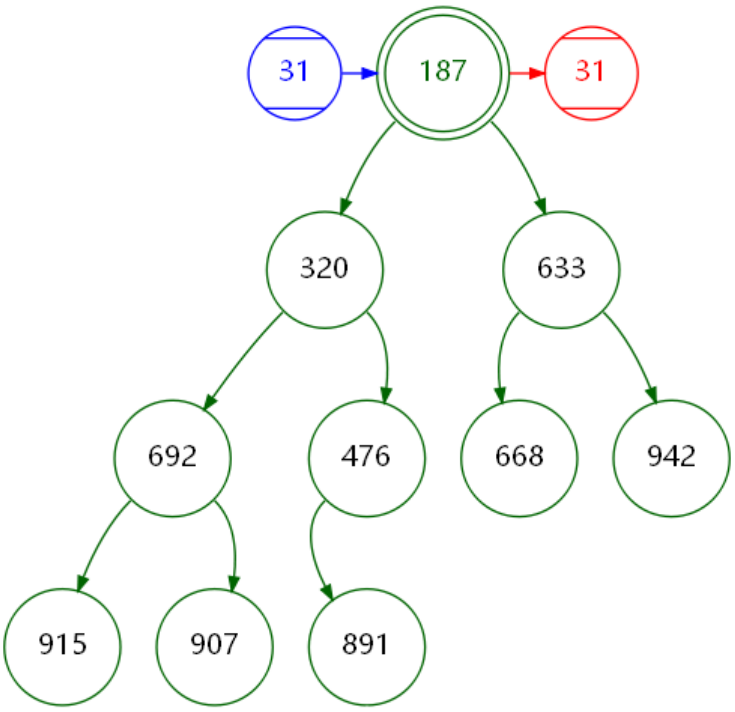
value		52	320	633	692	476	668	942	915	907	891
address	0	1	2	3	4	5	6	7	8	9	10

Time = 11



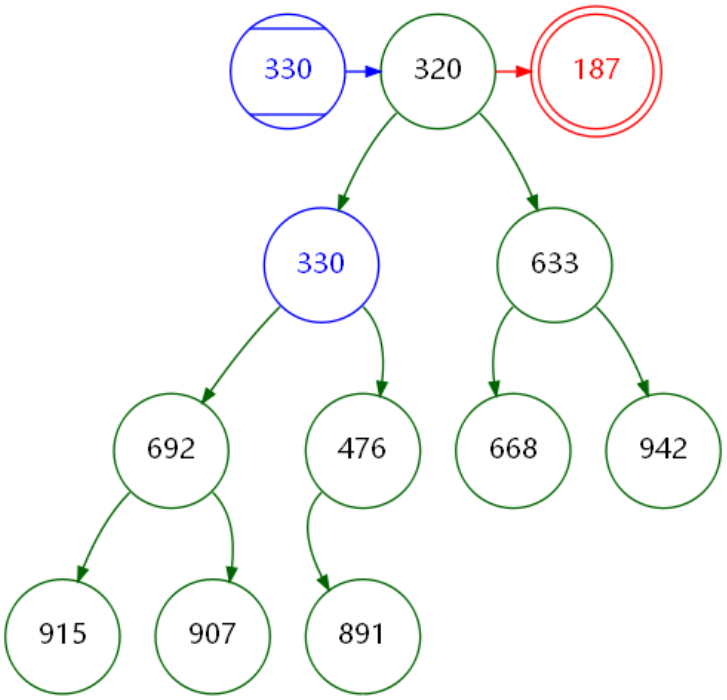
value		187	320	633	692	476	668	942	915	907	891
address	0	1	2	3	4	5	6	7	8	9	10

Time = 12



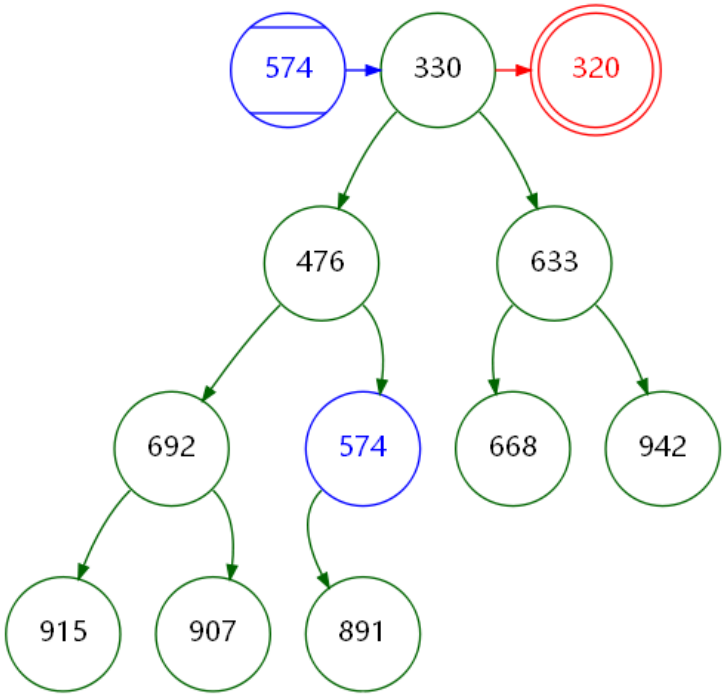
value		187	320	633	692	476	668	942	915	907	891
address	0	1	2	3	4	5	6	7	8	9	10

Time = 13



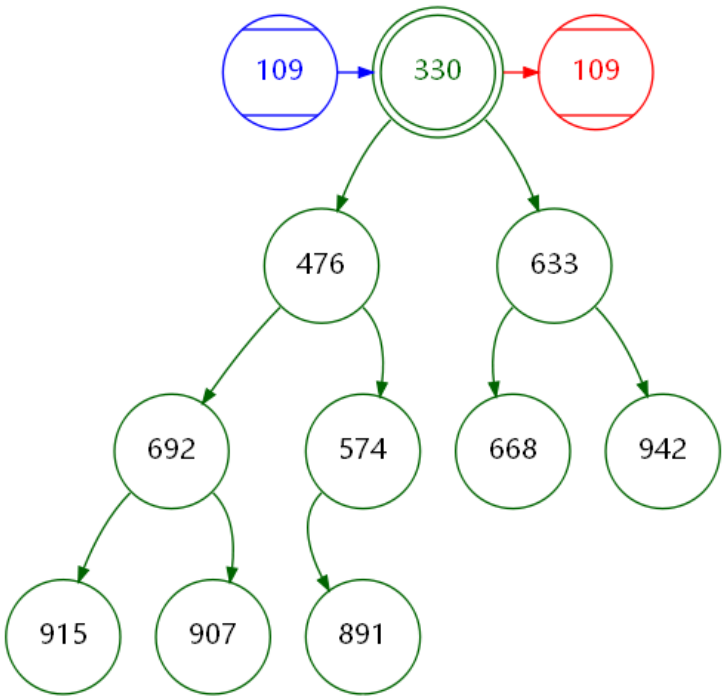
value		320	330	633	692	476	668	942	915	907	891
address	0	1	2	3	4	5	6	7	8	9	10

Time = 14



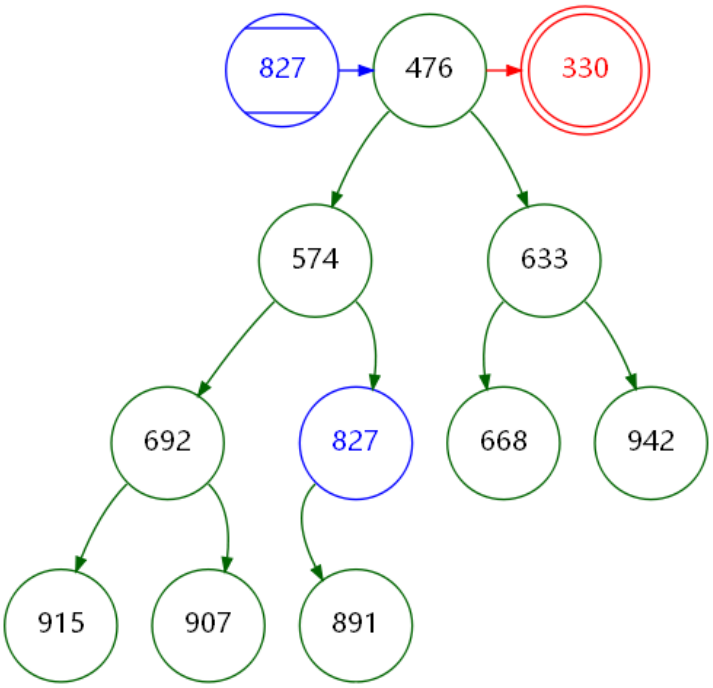
value		330	476	633	692	574	668	942	915	907	891
address	0	1	2	3	4	5	6	7	8	9	10

Time = 15



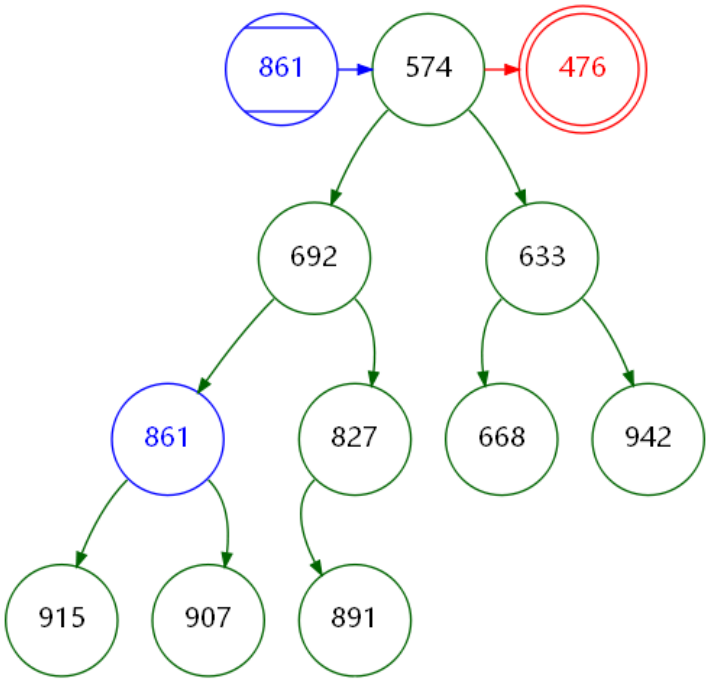
value		330	476	633	692	574	668	942	915	907	891
address	0	1	2	3	4	5	6	7	8	9	10

Time = 16



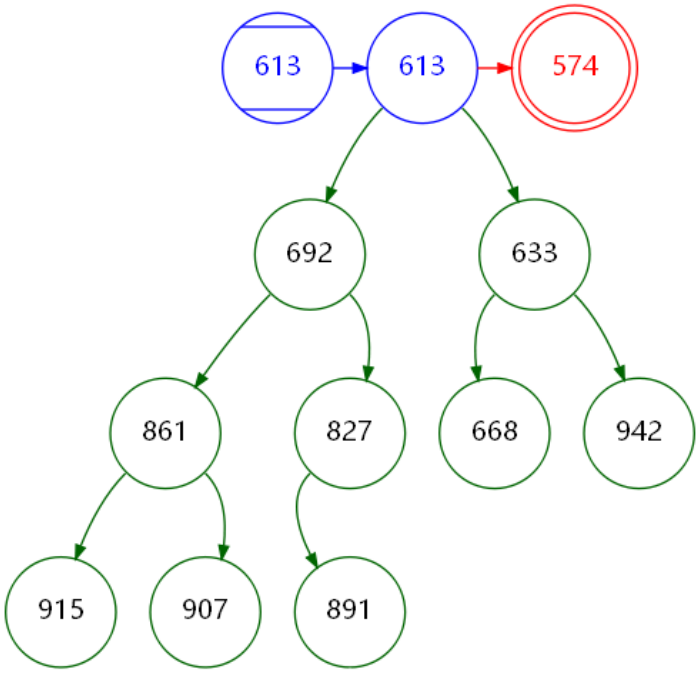
value		476	574	633	692	827	668	942	915	907	891
address	0	1	2	3	4	5	6	7	8	9	10

Time = 17



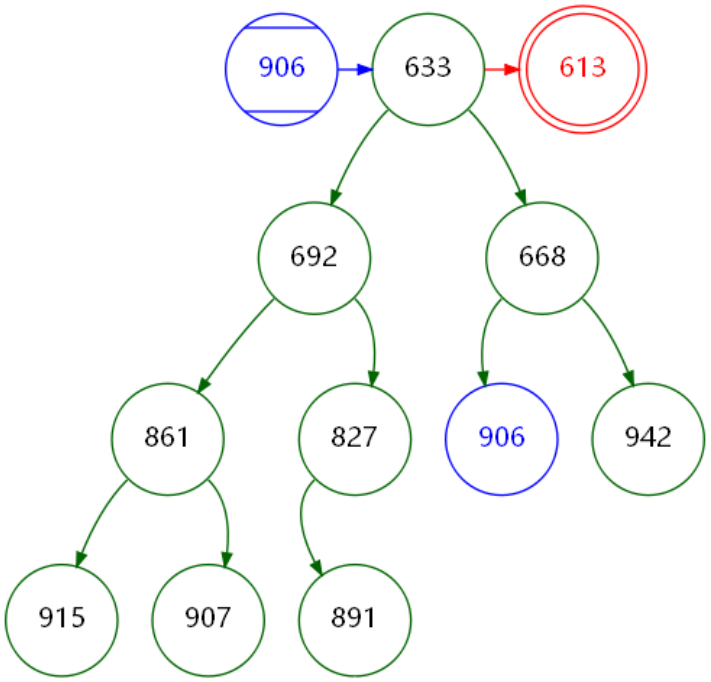
value		574	692	633	861	827	668	942	915	907	891
address	0	1	2	3	4	5	6	7	8	9	10

Time = 18



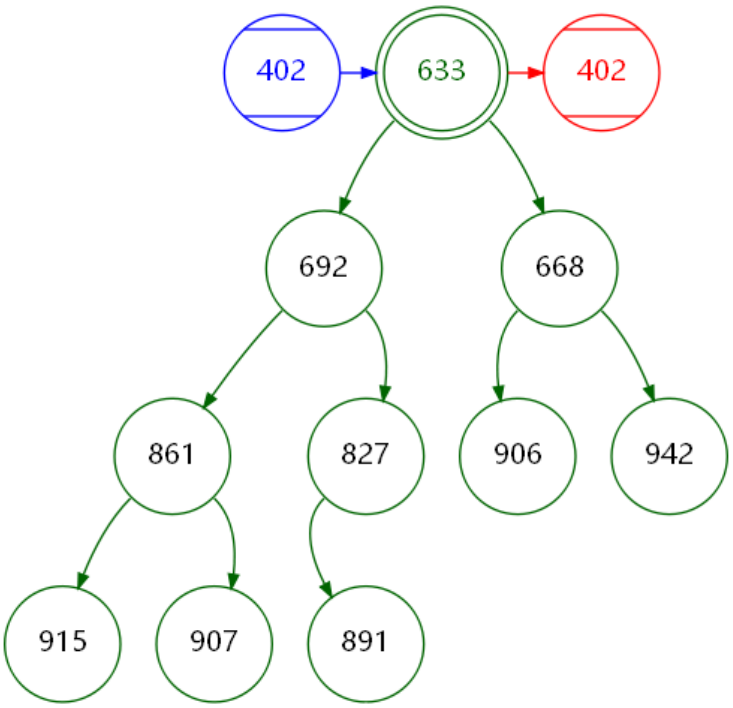
value		613	692	633	861	827	668	942	915	907	891
address	0	1	2	3	4	5	6	7	8	9	10

Time = 19



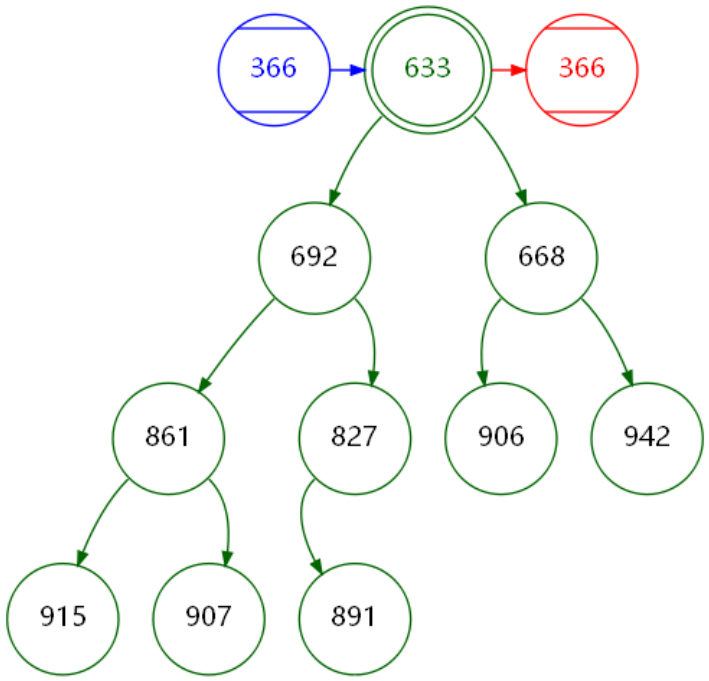
value		633	692	668	861	827	906	942	915	907	891
address	0	1	2	3	4	5	6	7	8	9	10

Time = 20



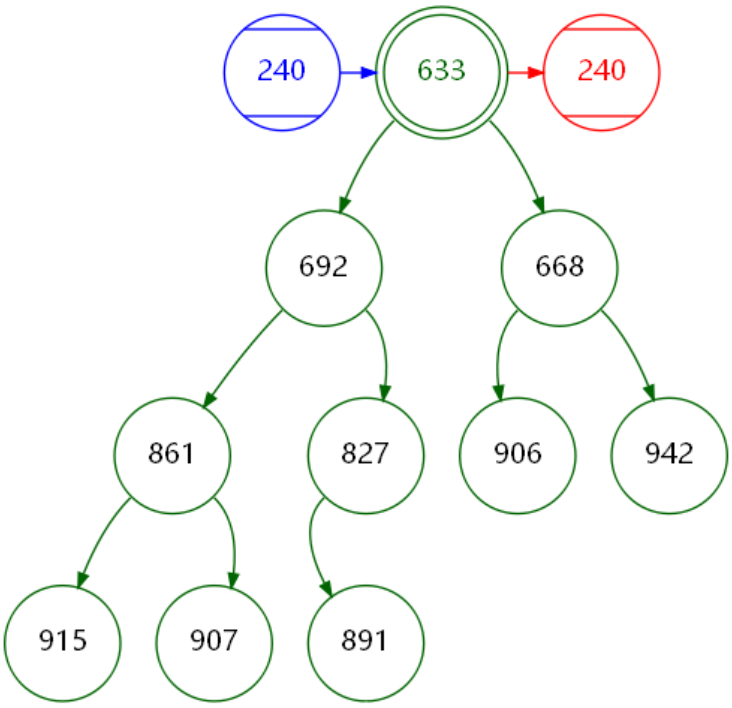
value		633	692	668	861	827	906	942	915	907	891
address	0	1	2	3	4	5	6	7	8	9	10

Time = 21



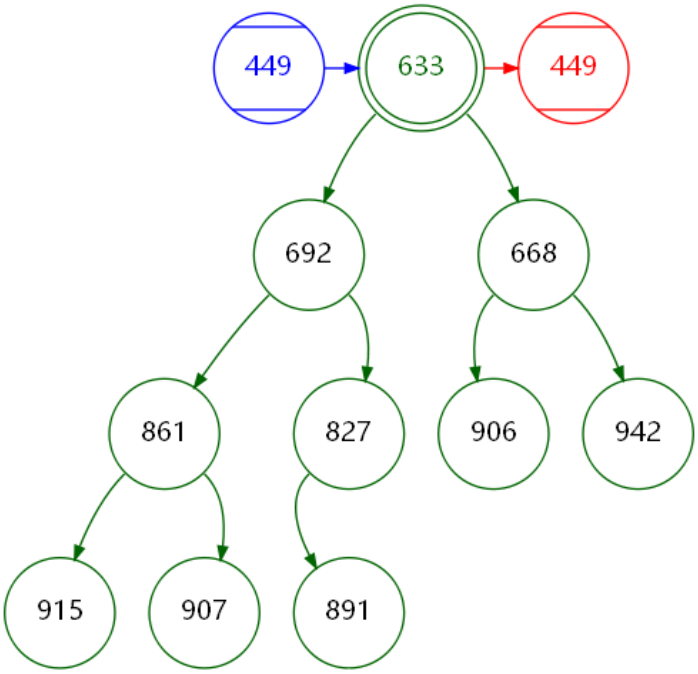
value		633	692	668	861	827	906	942	915	907	891
address	0	1	2	3	4	5	6	7	8	9	10

Time = 22



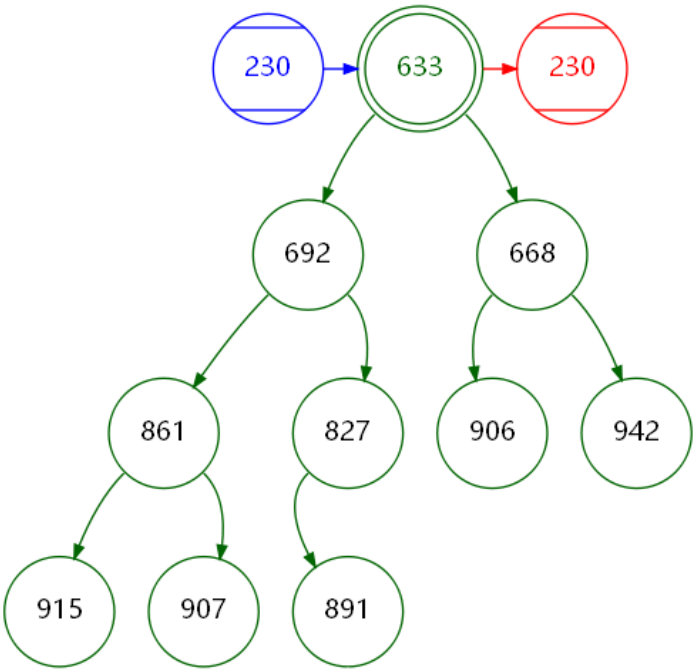
value		633	692	668	861	827	906	942	915	907	891
address	0	1	2	3	4	5	6	7	8	9	10

Time = 23



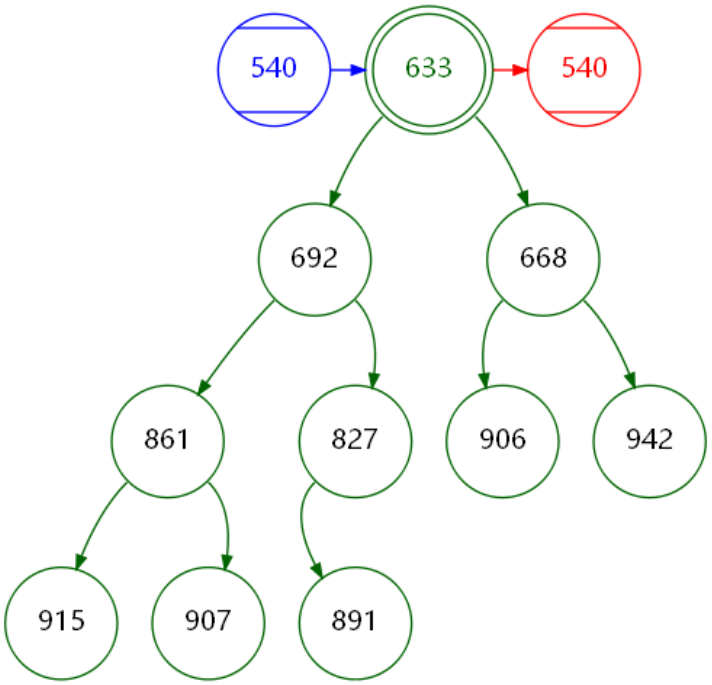
value		633	692	668	861	827	906	942	915	907	891
address	0	1	2	3	4	5	6	7	8	9	10

Time = 24



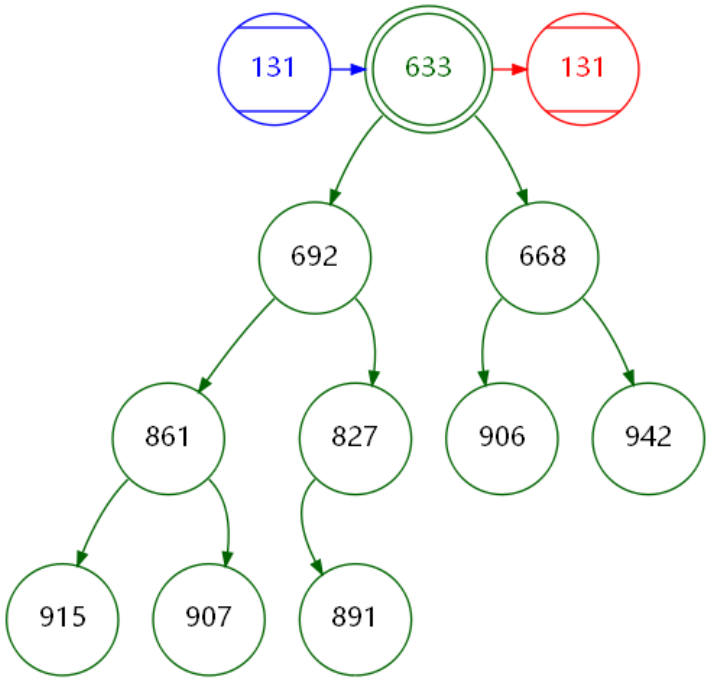
value		633	692	668	861	827	906	942	915	907	891
address	0	1	2	3	4	5	6	7	8	9	10

Time = 25



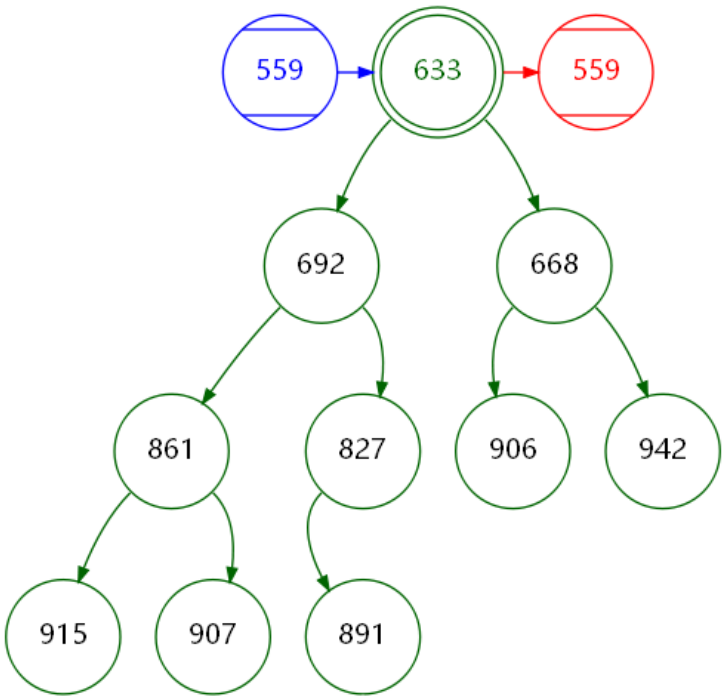
value		633	692	668	861	827	906	942	915	907	891
address	0	1	2	3	4	5	6	7	8	9	10

Time = 26



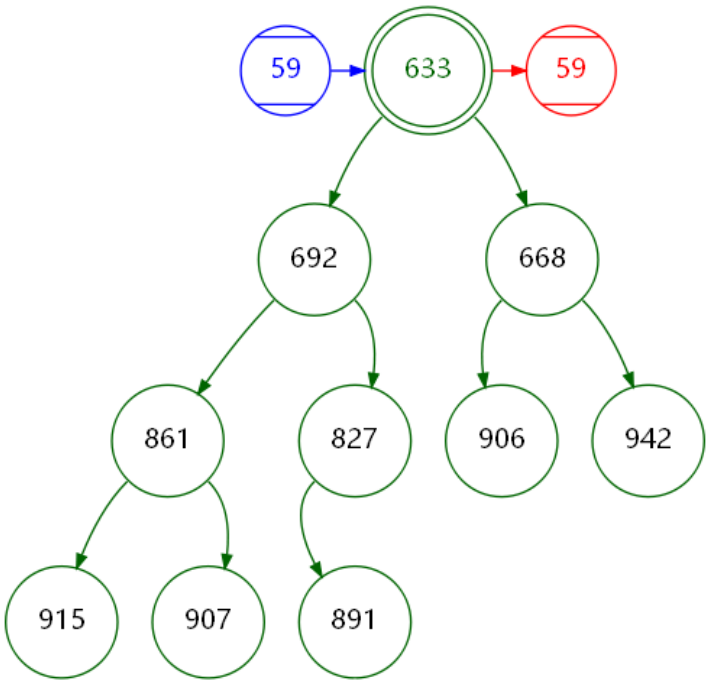
value		633	692	668	861	827	906	942	915	907	891
address	0	1	2	3	4	5	6	7	8	9	10

Time = 27



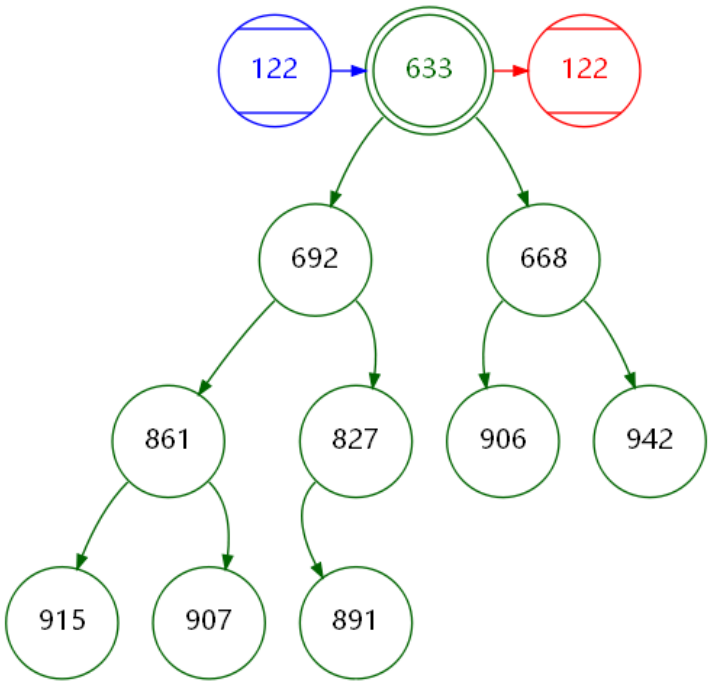
value		633	692	668	861	827	906	942	915	907	891
address	0	1	2	3	4	5	6	7	8	9	10

Time = 28



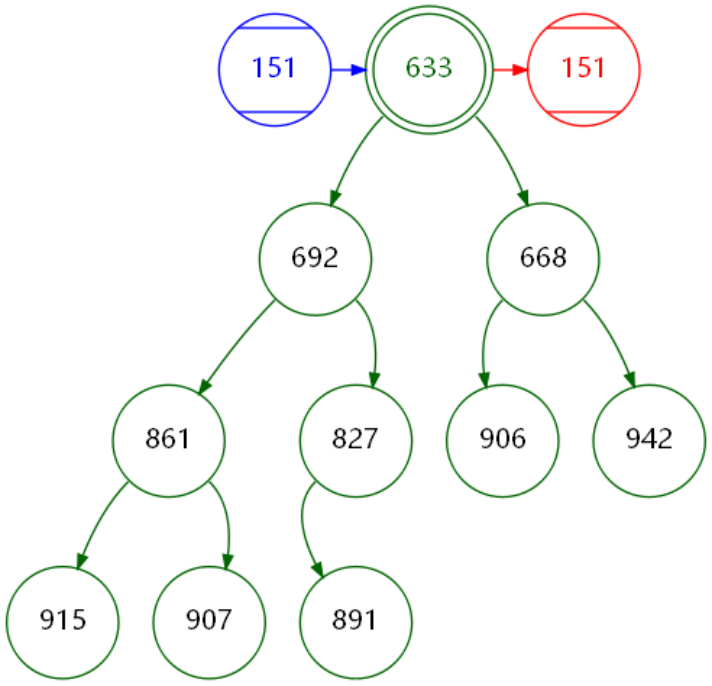
value		633	692	668	861	827	906	942	915	907	891
address	0	1	2	3	4	5	6	7	8	9	10

Time = 29



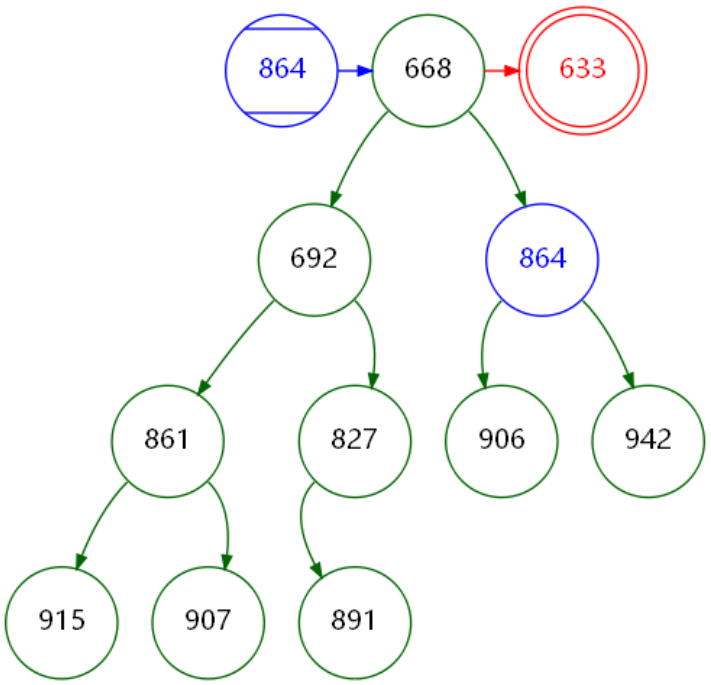
value		633	692	668	861	827	906	942	915	907	891
address	0	1	2	3	4	5	6	7	8	9	10

Time = 30



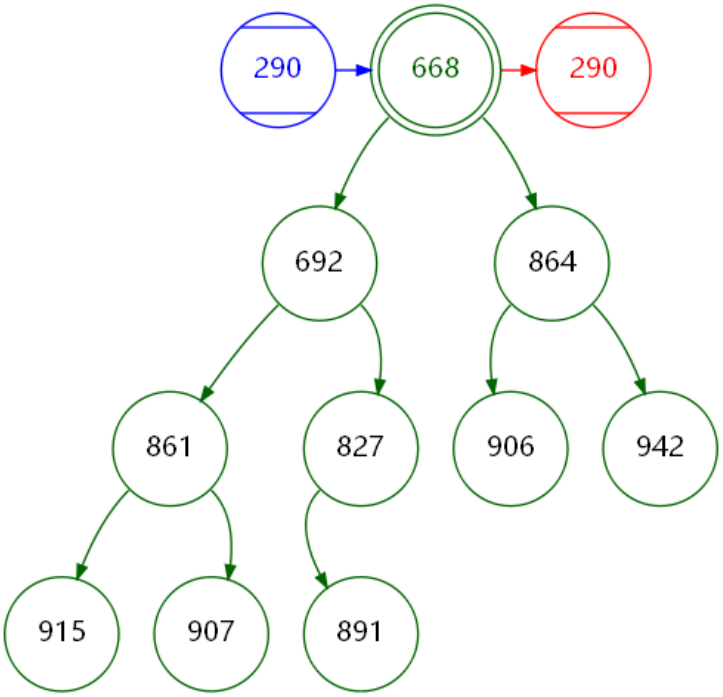
value		633	692	668	861	827	906	942	915	907	891
address	0	1	2	3	4	5	6	7	8	9	10

Time = 31



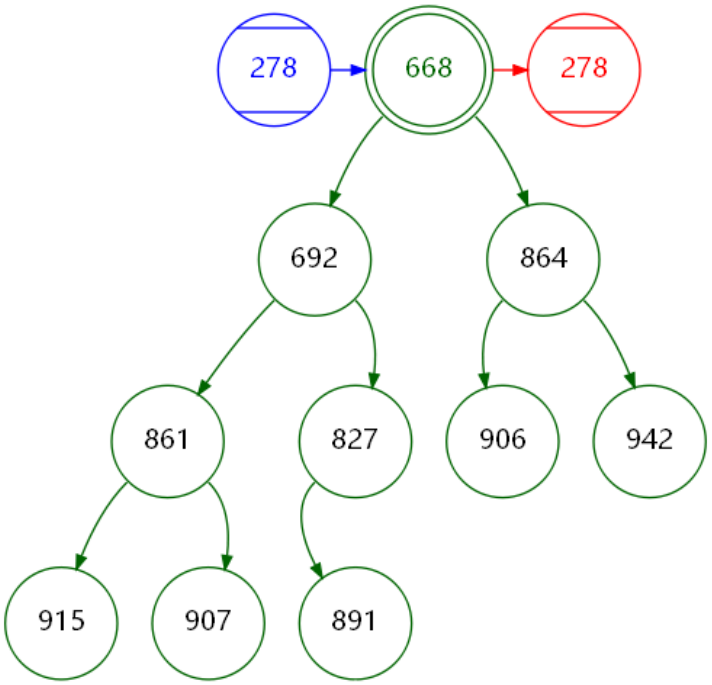
value		668	692	864	861	827	906	942	915	907	891
address	0	1	2	3	4	5	6	7	8	9	10

Time = 32



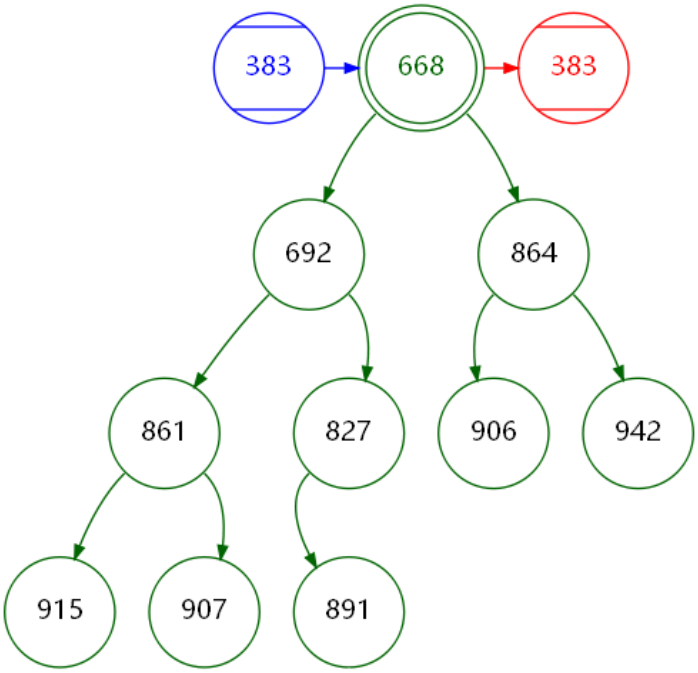
value		668	692	864	861	827	906	942	915	907	891
address	0	1	2	3	4	5	6	7	8	9	10

Time = 33



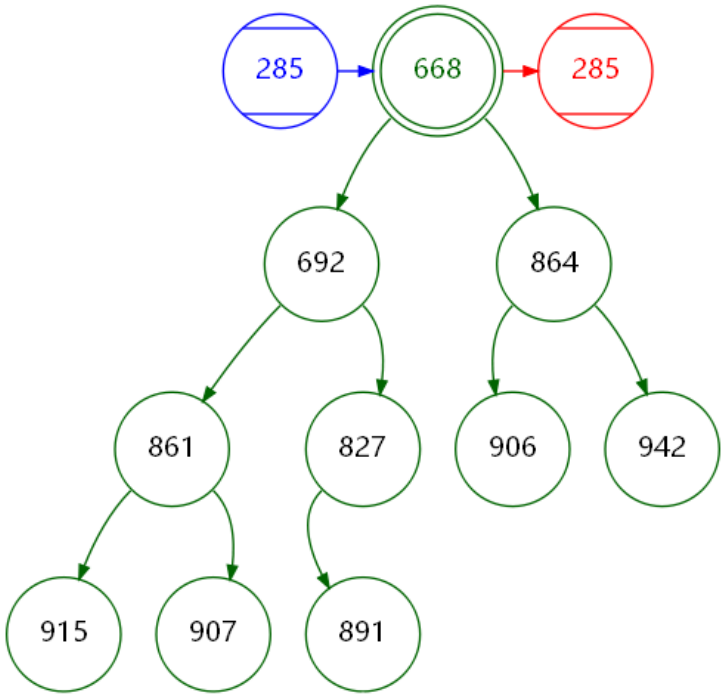
value		668	692	864	861	827	906	942	915	907	891
address	0	1	2	3	4	5	6	7	8	9	10

Time = 34



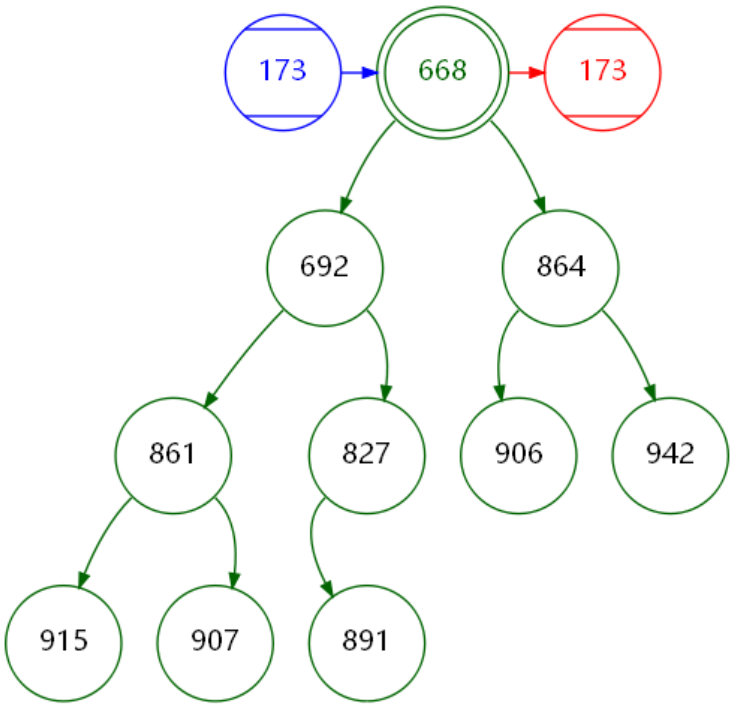
value		668	692	864	861	827	906	942	915	907	891
address	0	1	2	3	4	5	6	7	8	9	10

Time = 35



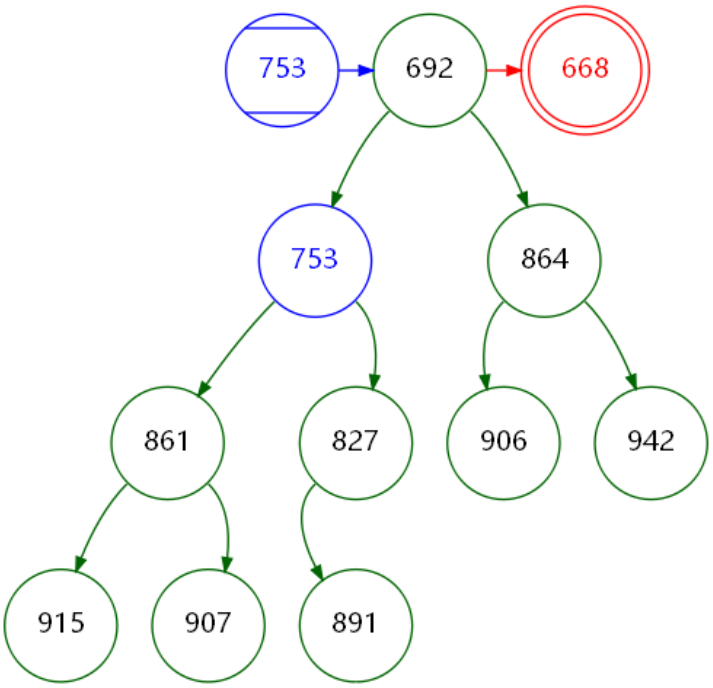
value		668	692	864	861	827	906	942	915	907	891
address	0	1	2	3	4	5	6	7	8	9	10

Time = 36



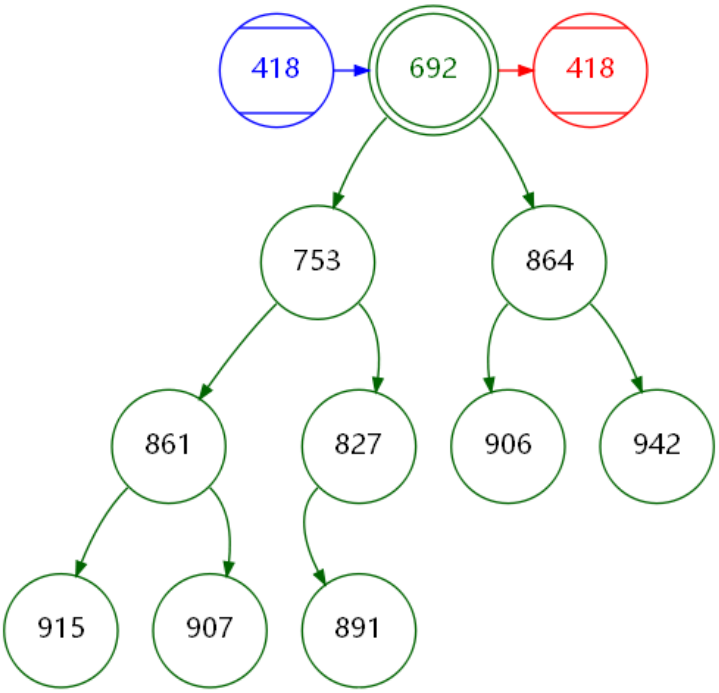
value		668	692	864	861	827	906	942	915	907	891
address	0	1	2	3	4	5	6	7	8	9	10

Time = 37



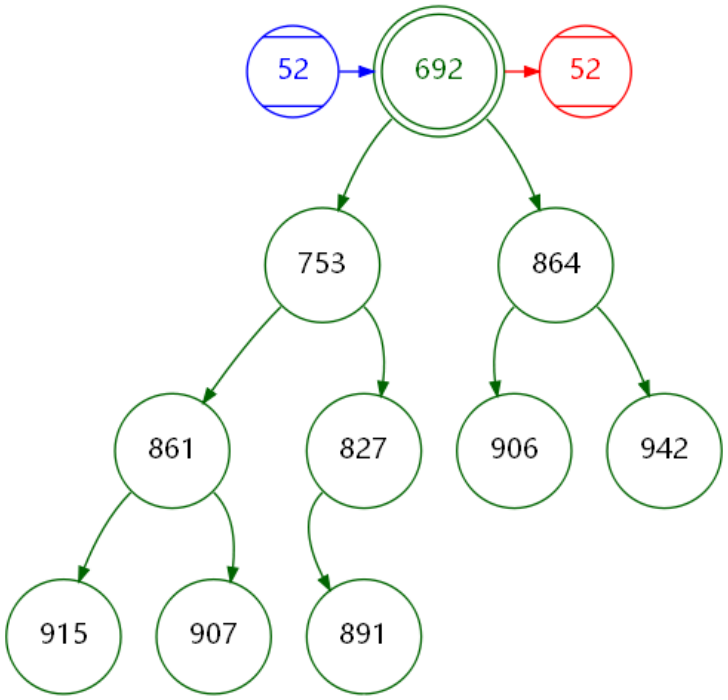
value		692	753	864	861	827	906	942	915	907	891
address	0	1	2	3	4	5	6	7	8	9	10

Time = 38



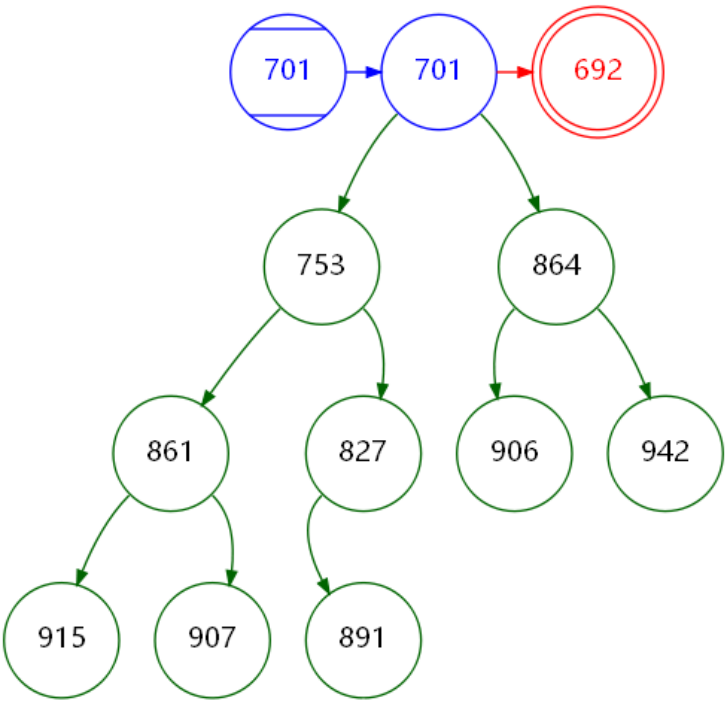
value		692	753	864	861	827	906	942	915	907	891
address	0	1	2	3	4	5	6	7	8	9	10

Time = 39

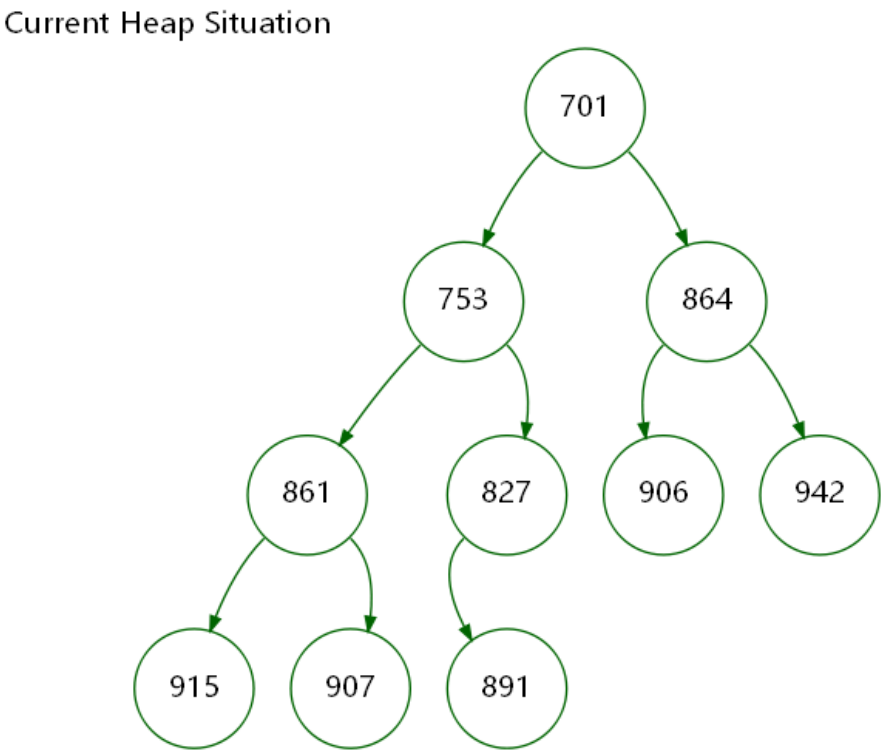


value		692	753	864	861	827	906	942	915	907	891
address	0	1	2	3	4	5	6	7	8	9	10

Time = 40



value		701	753	864	861	827	906	942	915	907	891
address	0	1	2	3	4	5	6	7	8	9	10



value		701	753	864	861	827	906	942	915	907	891
address	0	1	2	3	4	5	6	7	8	9	10

图 3 一次 Top10 的小顶堆调整过程

八、总结及心得体会：

本次实验实现了利用小顶堆的数据结构来解决 Top-K 问题，还实现了堆遍历、堆排序、带起点的堆调整等功能，因为可视化部分都体现了这些功能，所有都没有实际使用这些函数。在写小顶堆的时候参考了书上的代码，我 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，如果用 0 开始，不同的下标会导致父、子结点之间的计算方法不同，这个是一个需要注意的点。另外，我最初开始写堆排序的没有深刻理解堆排序的思想，它不是严格的排序算法，但是实现了排序的功能，真正理解堆排序后感觉豁然开朗。

九、对本实验过程及方法、手段的改进建议：

1. 在研究程序耗时 t 与数据量 N、小顶堆容量 k 的关系时，每组数据只测了 1 次，可以多次独立运行后取平均值，这样做的结果更加准确，误差也更小；

2. 加强 dot 语言的学习，改进可视化部分的代码，使生成的图片更加美观；

3. 更加完善代码注释，提高代码的可读性。

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 Dijkstra 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 Dijkstra 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™ i5-8300H CPU @ 2.30GHz　2.30GHz

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：Dev-C++ Version 5.11

编译器配置：TDM-GCC 4.9.2 64-bit Release

六、实验步骤

Dijkstra 算法

```
1. **********图的结构体定义**********
2. typedef struct graph
3. {
4.     int **edges;//邻接矩阵
5.     int n;//顶点数
6.     int e;//边数
7. }Graph;
8.
9. **********初始化部分**********
```

```
10. //初始化图
11. Graph CreateGraph(int n,int e)
12. {
13.     Graph graph;
14.     graph.n=n;
15.     graph.e=e;
16.     graph.edges=(int **)malloc(sizeof(int *)*graph.n);
17.
18.     for(int i=0;i<graph.n;i++)
19.     {
20.         graph.edges[i]=(int *)malloc(sizeof(int)*graph.n);
21.     }
22.
23.     for(int i=0;i<graph.n;i++)
24.     {
25.         for(int j=0;j<graph.n;j++)
26.         {
27.             graph.edges[i][j]=0;
28.         }
29.     }
30.
31.     return graph;
32. }
33.
34. //初始化 dist
35. int *CreateDist(int size)
36. {
37.     int *dist=new int[size];
38.     return dist;
39. }
40.
41. //初始化 path
42. int *CreatePath(int size)
43. {
44.     int *path=new int[size];
45.     return path;
46. }
47.
48. //初始化 visited
49. bool *CreateVisited(int size)
50. {
51.     bool *visited=(bool *)malloc(sizeof(bool)*size);
52.     return visited;
53. }
```

```
54.
55. //释放邻接矩阵空间
56. void FreeGraph(Graph g)
57. {
58.     for(int i=0;i<g.n;i++)
59.     {
60.         free(g.edges[i]);
61.     }
62.     free(g.edges);
63. }
64.
65. ********************打印部分********************
66. //打印源顶点 vs 到各结点的最短路径
67. void PrintPath(Graph g,int *dist,int *path,int vs)
68. {
69.     for(int i=0;i<g.n;i++)
70.     {
71.         if(vs!=i)
72.         {
73.             printf("v%d -> v%d, minDist: %d, path: v%d <- ",vs,i,
    dist[i],i);
74.             int temp=path[i];
75.             while(vs!=temp)
76.             {
77.                 printf("v%d <- ",temp);
78.                 temp=path[temp];
79.             }
80.             printf("v%d",vs);
81.             printf("\n");
82.         }
83.     }
84. }
85.
86. //打印邻接矩阵
87. void PrintGraph(Graph g)
88. {
89.     for(int i=0;i<g.n;i++)
90.     {
91.         for(int j=0;j<g.n;j++)
92.         {
93.             printf("%2d ",g.edges[i][j]);
94.         }
95.         printf("\n");
96.     }
```

```
97. }
98.
99. ********************可视化部分********************
100.    void DijkstraDot(Graph g,int *path,bool *visited,int vs)
101.    {
102.        FILE *fp=fopen("dijkstra.gv","w+");
103.        fprintf(fp,"digraph Dijkstra {\nnode [shape=ellipse];\n")
    ;
104.        fprintf(fp,"v%d[shape=diamond,color=red,fontcolor=red];\n
    ",vs);
105.
106.        for(int i=0;i<g.n && i!=vs;i++)
107.        {
108.            fprintf(fp,"v%d;\n",i);
109.        }
110.
111.        for(int i=0;i<g.n;i++)
112.        {
113.            for(int j=0;j<g.n;j++)
114.            {
115.                if(g.edges[i][j])
116.                {
117.                    if(visited[i] && visited[j] && path[j]==i)
118.                    {
119.                        fprintf(fp,"v%d[fontcolor=red,color=red];
    \n",i);
120.                        fprintf(fp,"v%d[fontcolor=red,color=red];
    \n",j);
121.                        fprintf(fp,"v%d->v%d[style=bold,label=%d,
    fontcolor=red,color=red];\n",i,j,g.edges[i][j]);
122.                    }
123.                    else
124.                    {
125.                        fprintf(fp,"v%d->v%d[style=bold,label=%d]
    ;\n",i,j,g.edges[i][j]);
126.                    }
127.                }
128.            }
129.        }
130.        fprintf(fp,"}\n");
131.        fclose(fp);
132.    }
133.
134.    //vs 表示源顶点
```

```
135.    void DijkstraPath(Graph g,int *dist,int *path,int vs)
136.    {
137.        bool *visited=CreateVisited(g.n);
138.        //初始化
139.        for(int i=0;i<g.n;i++)
140.        {
141.            if(g.edges[vs][i]>0 && i!=vs)
142.            {
143.                dist[i]=g.edges[vs][i];
144.                //path 记录最短路径上从 vs 到 i 的前一个顶点
145.                path[i]=vs;
146.            }
147.            else
148.            {
149.                //若 i 不与 vs 直接相邻，则权值置为无穷大
150.                dist[i]=INT_MAX;
151.                path[i]=-1;
152.            }
153.            visited[i]=false;
154.            path[vs]=vs;
155.            dist[vs]=0;
156.        }
157.        FILE *fp=fopen("dijkstra.gv","w+");
158.        fprintf(fp,"digraph Dijkstra {\nnode [shape=ellipse];\n")
    ;
159.        fprintf(fp,"v%d[shape=diamond,color=red,fontcolor=red];\n
    ",vs);
160.
161.        for(int i=0;i<g.n && i!=vs;i++)
162.        {
163.            fprintf(fp,"v%d; ",i);
164.        }
165.
166.        for(int i=0;i<g.n;i++)
167.        {
168.            for(int j=0;j<g.n;j++)
169.            {
170.                if(g.edges[i][j])
171.                {
172.                    fprintf(fp,"v%d->v%d[style=bold,label=%d];\n"
    ,i,j,g.edges[i][j]);
173.                }
174.            }
175.        }
```

```
176.         fprintf(fp,"}\n");
177.         fclose(fp);
178.         system("sfdp.exe -Tpng dijkstra.gv -o DijkSetp01.png");
179.         visited[vs]=true;
180.         //循环扩展 n-1 次
181.         for(int i=1;i<g.n;i++)
182.         {
183.             int min=INT_MAX;
184.             int u;
185.             //寻找未被扩展的权值最小的顶点
186.             for(int j=0;j<g.n;j++)
187.             {
188.                 if(!visited[j] && dist[j]<min)
189.                 {
190.                     min=dist[j];
191.                     u=j;
192.                 }
193.             }
194.             visited[u]=true;
195.             //更新 dist 数组的值和路径的值
196.             for(int k=0;k<g.n;k++)
197.             {
198.                 if(!visited[k] && g.edges[u][k]>0 && min+g.edges[
    u][k]<dist[k])
199.                 {
200.                     dist[k]=min+g.edges[u][k];
201.                     path[k]=u;
202.                 }
203.             }
204.             DijkstraDot(g,path,visited,vs);
205.             char orderstr[128];
206.             sprintf(orderstr,"sfdp.exe -Tpng dijkstra.gv -o DijkS
    etp%02d.png",i+1);
207.             system(orderstr);
208.         }
209. }
```

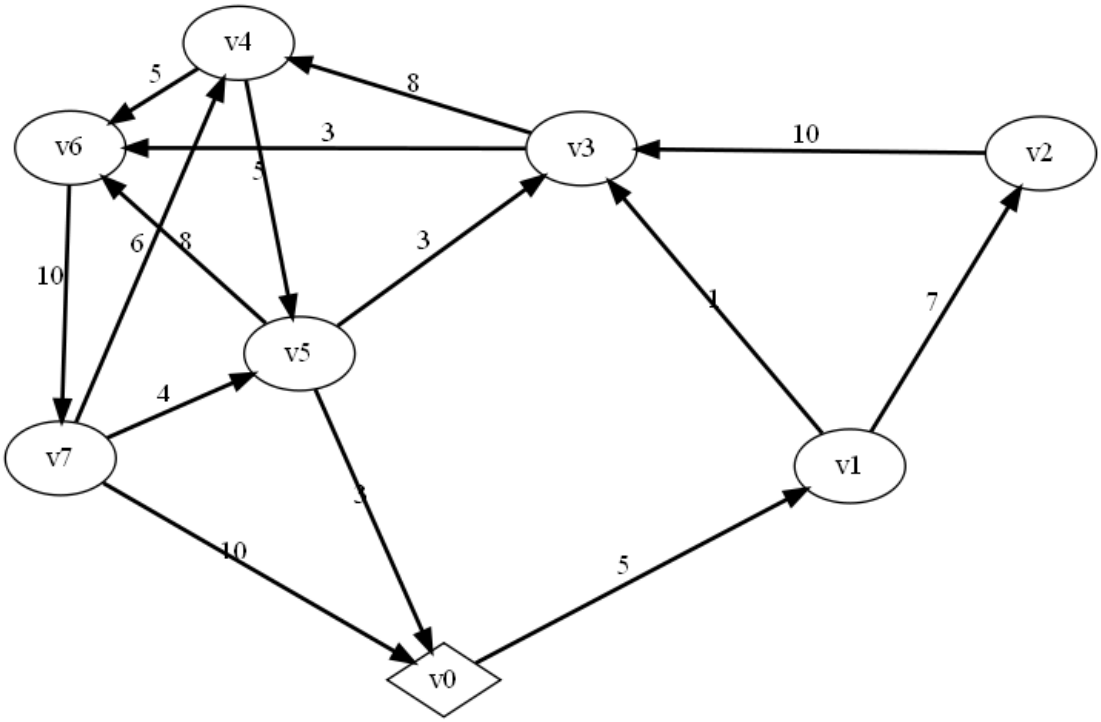
七、实验数据及结果分析

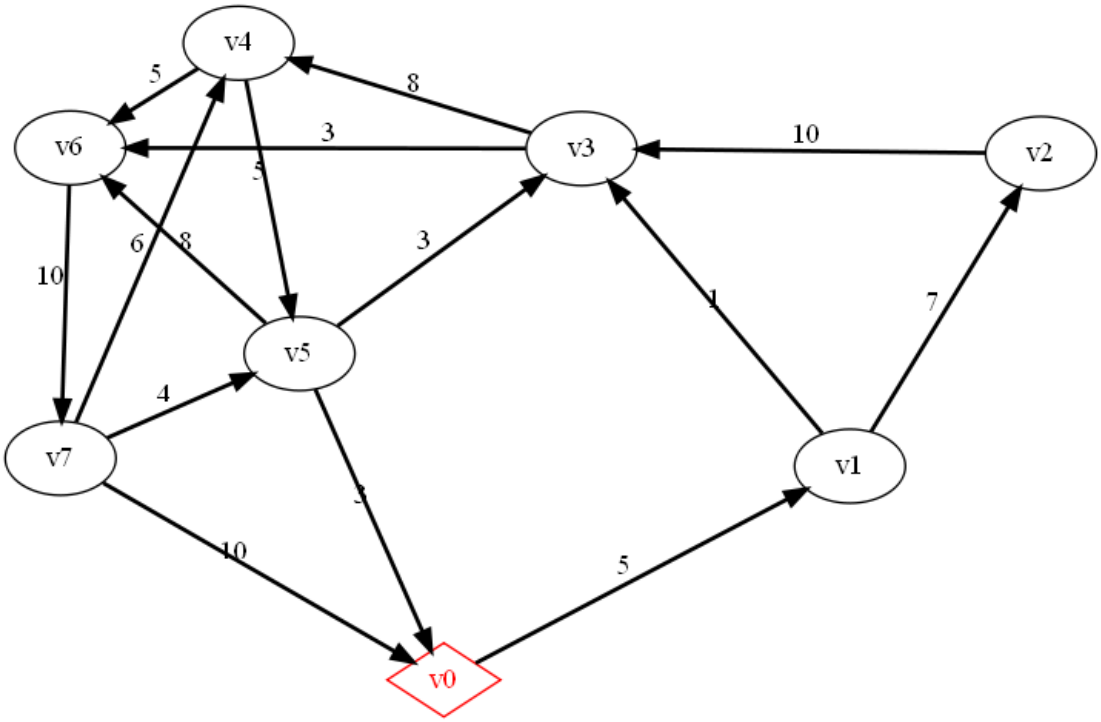
运行程序，创建一个 8 个结点、15 条边的有向图 G，随机生成的邻接矩阵如图 1 所示。

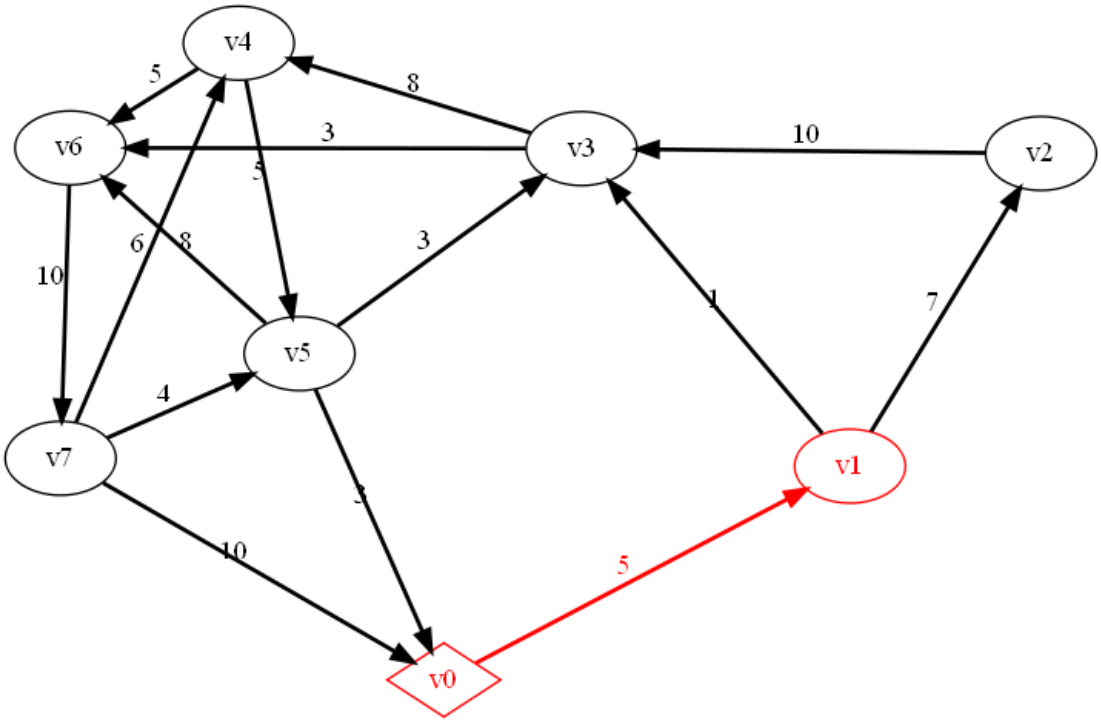
0	5	0	0	0	0	0	0
0	0	7	1	0	0	0	0
0	0	0	10	0	0	0	0
0	0	0	0	8	0	3	0
0	0	0	0	0	5	5	0
3	0	0	3	0	0	8	0
0	0	0	0	0	0	0	10
10	0	0	0	6	4	0	0

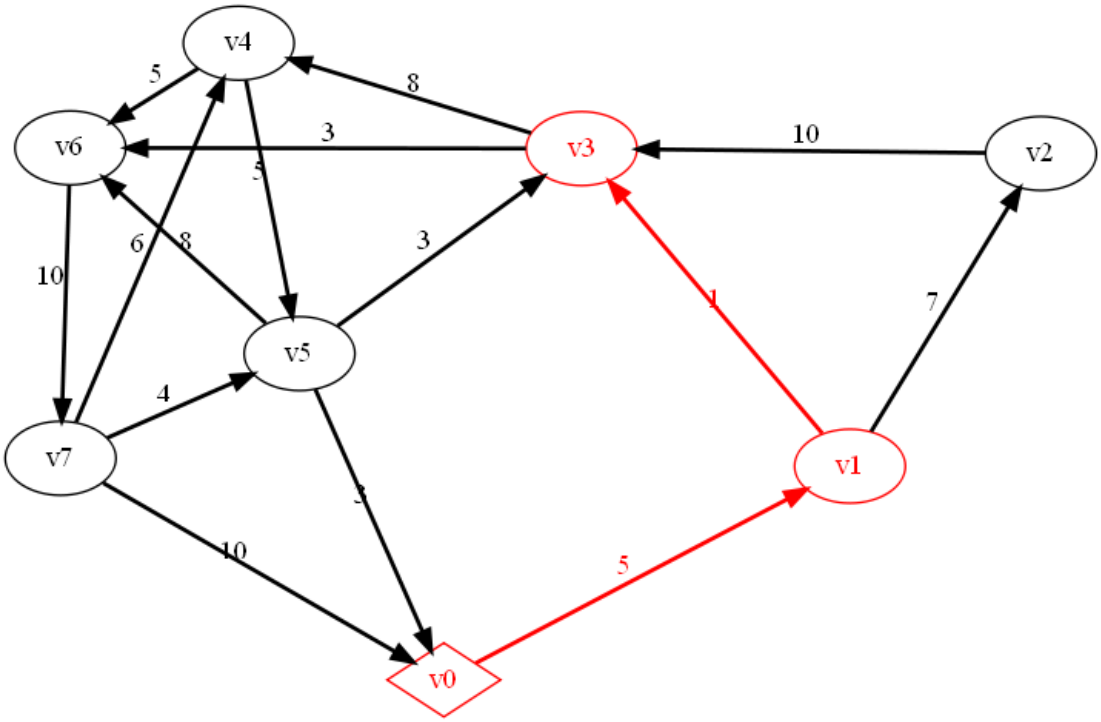
图 1　图 G 的邻接矩阵

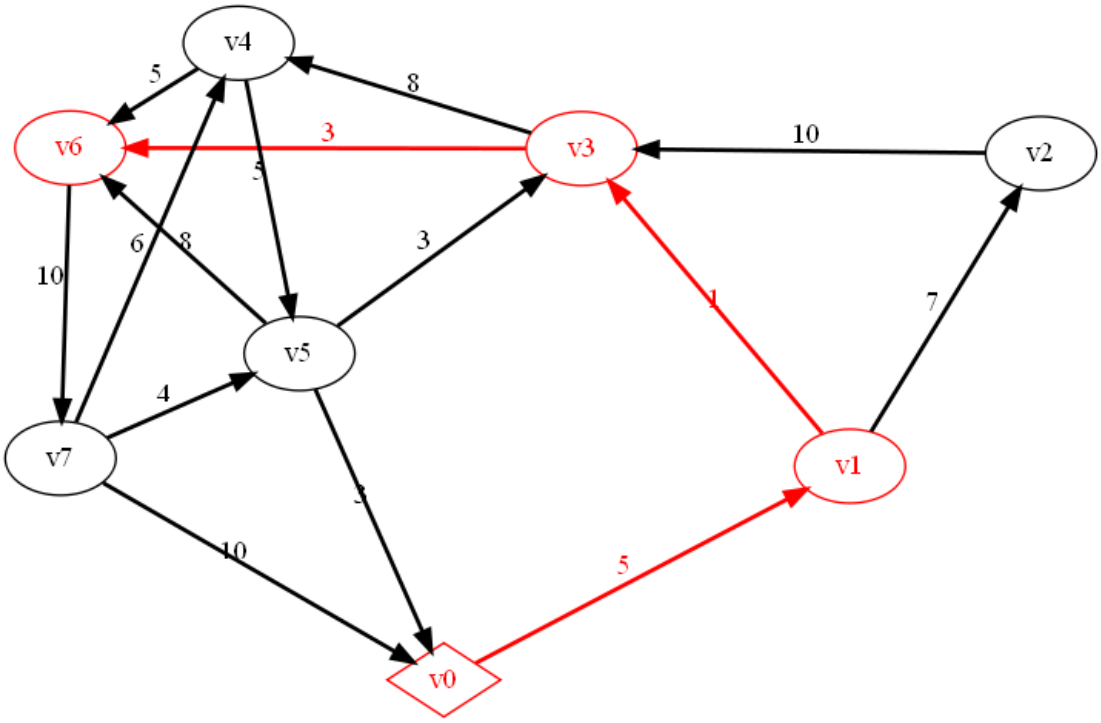
图 G 的最短路径算法过程如图 2 所示。

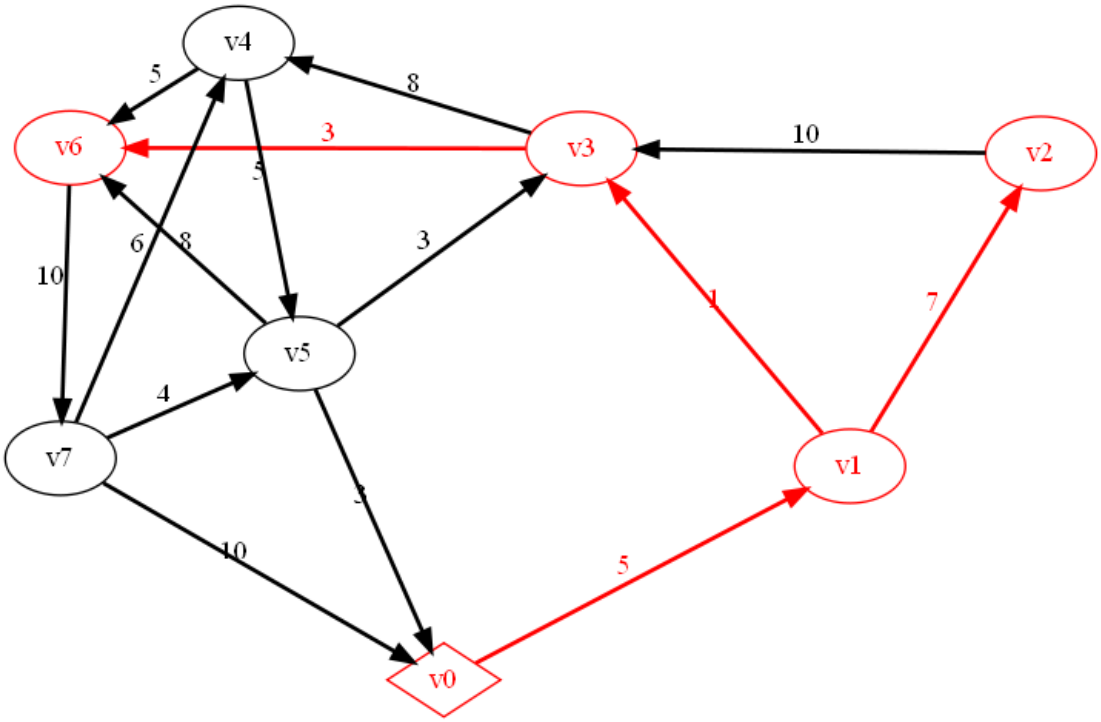


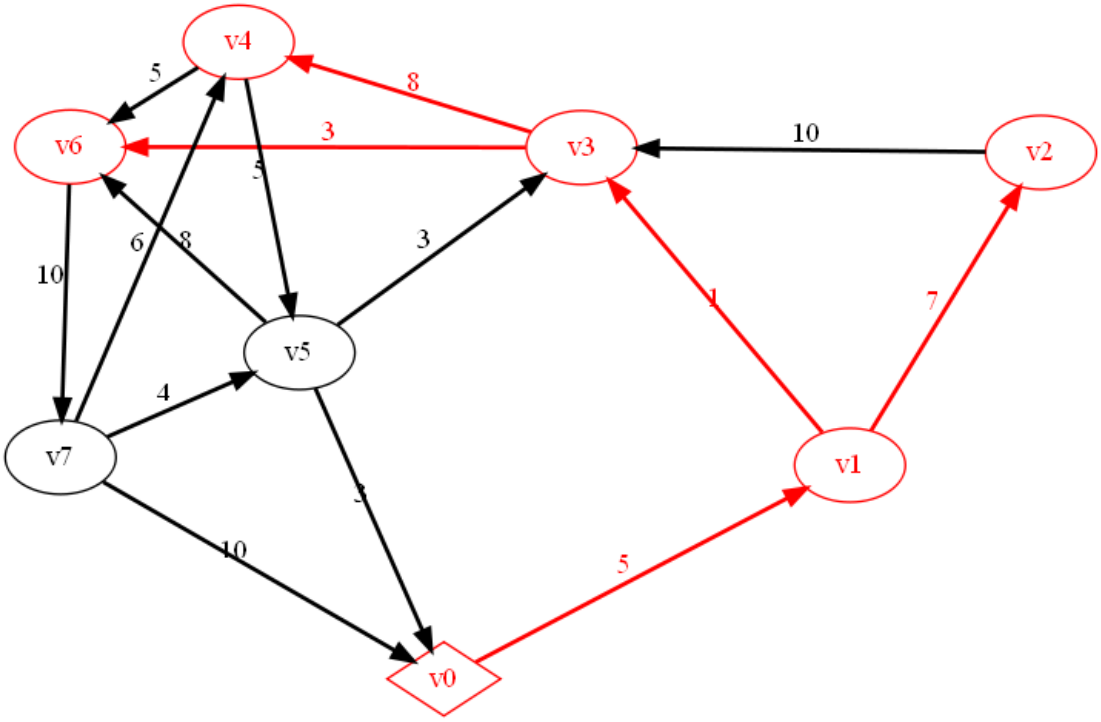


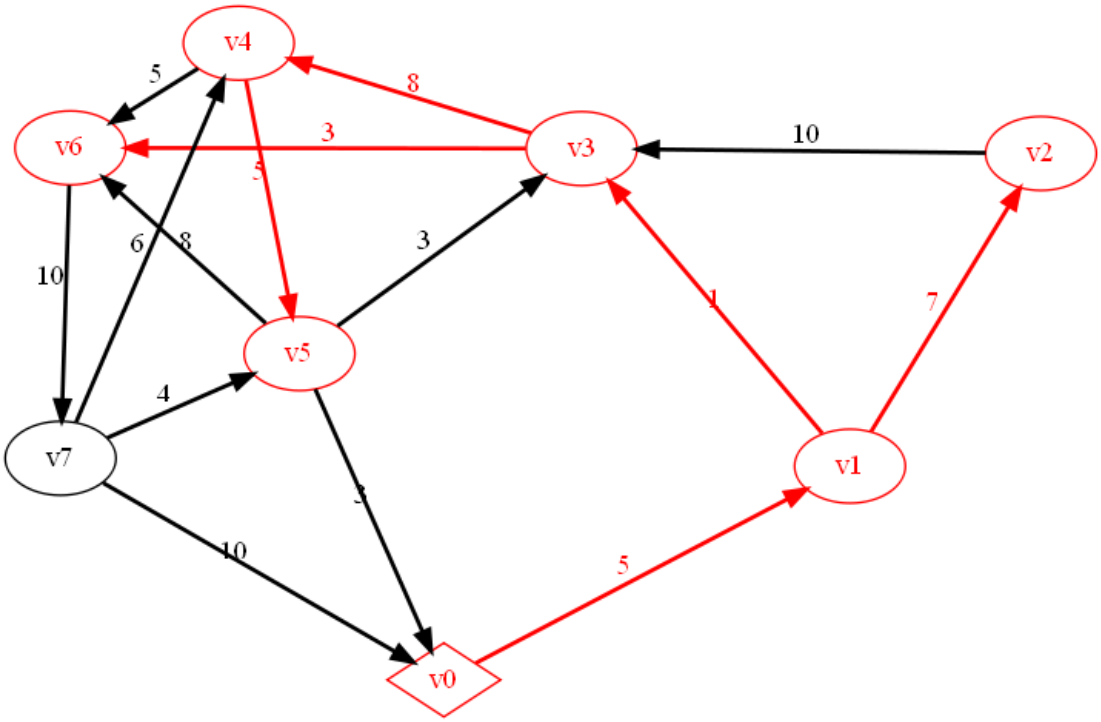

v4
v6
v3
v2
v5
v7
v1
v0
5
8
3
10
6
8
3
10
1
7
4
10
5


v4
v6
v3
v2
v5
v7
v1
v0
5
8
3
10
6
8
3
10
1
7
4
10
5


v4
v6
v3
v2
v5
v7
v1
v0
5
8
3
10
6
8
3
10
1
7
4
10
5


v4
v6
v3
v2
v5
v7
v1
v0
5
8
3
10
6
8
3
10
1
7
4
10
5


v4
v6
v3
v2
v5
v7
v1
v0
5
8
3
10
6
8
3
10
1
7
4
10
5


v4
v6
v3
v2
v5
v7
v1
v0
5
8
3
10
6
8
3
10
1
7
4
10
5

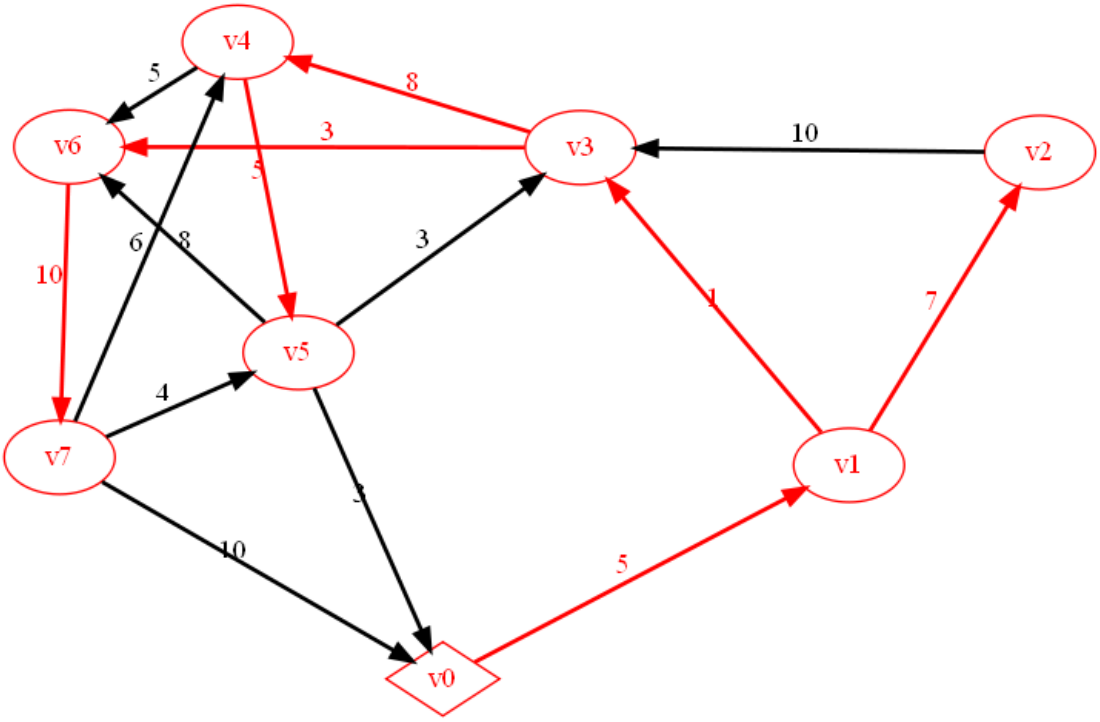


图 2　图 G 的最短路径算法过程

在图 G 中，源顶点 v0 到各个结点的最短路径如图 3 所示。

```
源顶点v0到各结点的最短路径为:
v0 -> v1, minDist: 5, path: v1 <- v0
v0 -> v2, minDist: 12, path: v2 <- v1 <- v0
v0 -> v3, minDist: 6, path: v3 <- v1 <- v0
v0 -> v4, minDist: 14, path: v4 <- v3 <- v1 <- v0
v0 -> v5, minDist: 19, path: v5 <- v4 <- v3 <- v1 <- v0
v0 -> v6, minDist: 9, path: v6 <- v3 <- v1 <- v0
v0 -> v7, minDist: 19, path: v7 <- v6 <- v3 <- v1 <- v0
```

图 3 源顶点 v0 到各个结点的最短路径

八、总结及心得体会：

本次实验实现了 Dijkstra 算法、打印邻接矩阵、打印源顶点到各个结点的最短路径等功能。在生成邻接矩阵的过程中使用了随机数，导致有向图的形状是随机的。若结点数和边数设定的数值过高，有时会出现看不清边权的情况，导致可读性较差。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，我没有仔细考虑可视化的问题，导致一些函数的封装性较差，这也是有待改进的地方。

九、对本实验过程及方法、手段的改进建议：

1. 完善可视化部分的代码，将源顶点 v0 到各个结点的最短路径嵌入图片，

使生成的图片更加具有可读性；

2. 加强 dot 语言的学习，解决有时看不清边权的问题，使生成的图片更加美观；

3. 更加完善代码注释，提高代码的可读性。