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（计科专业）

学生姓名：岳子豪

学　　号：**2018051404015**

指导教师：周益民

日　　期：**2020 年 12 月 9 日**

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。从根节点开始，迭代或者递归向下移动，直到遇到一个空的指针，即把需要插入的值存储在该位置。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。由于该操作是构建二叉排序树最基本的操作，因此可以通过设计函数 InsertBST(BSTree *bst, KeyType key)来实现。

(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。

平衡二叉树的实现

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。

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。

五、实验器材（设备、元器件）

处理器：	Intel® Core™ i5-7200U CPU @ 2.50GHz 2.70GHz
已安装的内存(RAM)：	4.00GB (3.81GB 可用)
系统类型：	64 位操作系统，基于 x64 的处理器

编程语言：	C++ 17
IDE：	Jetbrains CLion 2019.2.5
环境：	MinGW-W64 GCC-8.1.0
编译器：	g++ (i686-win32-dwarf-rev0) 8.1.0
调试器：	GNU gdb (GDB) 8.1
可视化：	Graphviz 2.39

六、实验步骤

二叉排序树的实现

(1) BST 数据结构定义

二叉排序树的构建需要树型结构，因此 BST 的数据结构需包含树结构的属性，包括值（key）、左孩子指针域（*lchild）和右孩子指针域（*rchild），由于需要对树进行可视化，还需要一个辅助标志变量（flag）。结构定义代码如下：

```
typedef int KeyType;
```

```
typedef struct  node
{
       KeyType  key ;
       struct node  *lchild,*rchild;
       int flag;
}BSTNode, *BSTree;
```

(2) BST 构建流程

如何构建 BST 是本实验的核心问题，从 BST 的定义和性质出发，我们把 BST 的构建看成是先建立一个根节点，从根节点开始逐渐将所有元素插入，进而将问题转移到“如何向 BST 中插入节点”上。本实验中我们从文件中读取数据。对应代码比较简单，如下：

```
void  CreateBST(BSTree  *bst, char * filename)
{
       FILE *fp;
       KeyType keynumber;
       *bst=NULL;
       fp = fopen(filename,"r+");
       if(fp==NULL)
                exit(0x01);
       while(EOF != fscanf(fp,"%d",&keynumber))
                InsertBST(bst, keynumber);
}
```

(3) BST 节点插入算法

向 BST 中节点是构建 BST 最核心、最基本的操作，直接决定生成 BST 的可行性和正确性。根据 BST 的性质，根据实验原理中的分析，设计 BST 节点插入算法如下：

- 若当前二叉查找树为空，则插入元素作为根节点；
- 若不为空，且插入元素值小于当前节点值，则插入左子树；
- 若不为空，且插入元素值大于当前节点值，则插入右子树；

上述算法通过递归调用，实现对最终要插入的地方的父节点的访问，并完成最后一次递归调用，将新元素插入 BST 中。该算法的流程图如下图所示：

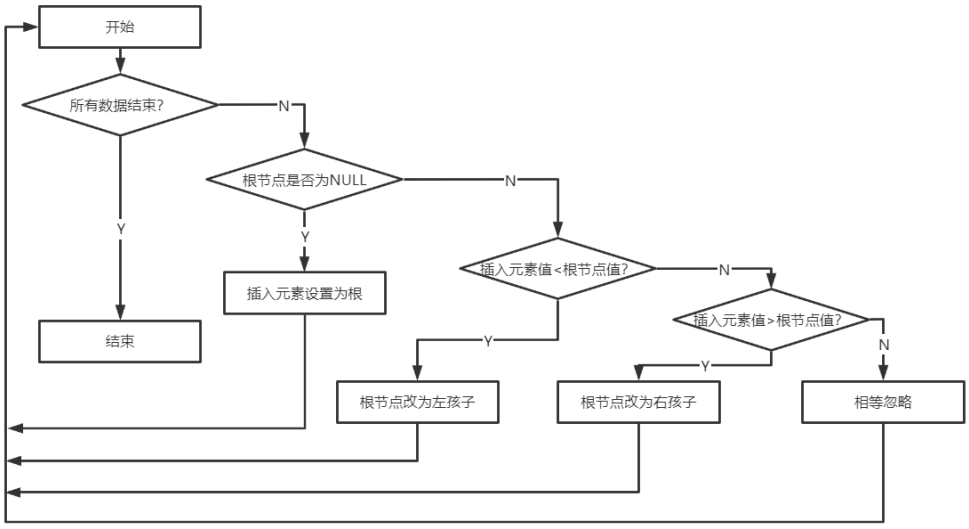


图 1 BST 节点插入算法流程图

对应代码如下：

```
void InsertBST(BSTree *bst, KeyType key)
{
	BSTree s;
	if (*bst == NULL)/*递归结束条件*/
	{
		s=(BSTree)malloc(sizeof(BSTNode));/*申请新的结点 s*/
		s-> key=key;
		s->lchild=NULL;
		s->rchild=NULL;
		s->flag=0;
		*bst=s;
	}
	else
		if (key < (*bst)->key)
			InsertBST(&((*bst)->lchild), key);/*将 s 插入左子树*/
		else
			if (key > (*bst)->key)
				InsertBST(&((*bst)->rchild), key); /*将 s
插入右子树*/
}
```

(4) BST 节点删除

作为本实验的附加功能，在 BST 中删除给定值节点相较于节点的插入麻烦很多。BST 节点删除包括查找和删除两大步，其中，查找算法基于 BST 结构，从根节点开始，不断比对查找值与根节点（包含子树根节点）的大小，判断是否找到或是否应该左子树或右子树，循环此过程直到找到查找值或遍历完成，使用基

于循环的非递归算法实现。查找功能的代码段如下：

```
BSTNode  *p, *f,*s ,*q;
p=t;
f=NULL;
while(p)
{
        if(p->key==k )  break;
        f=p;
        if(p->key>k)
                p=p->lchild;
        else
                p=p->rchild;
}
if(p==NULL)  return t;
```

删除操作在查找命中之后，涉及到三种可能的情况：

- 命中节点无子树，直接删除之；
- 命中节点度为 1，以其非空子节点替代之；
- 命中节点度为 2 的节点，以其前驱节点替代之。

具体算法实现如下（假设已有指针 p 命中带查找的节点）：

```
if(p->lchild==NULL)
{
        if(f==NULL)
                t=p->rchild;
        else
                if(f->lchild==p)
                        f->lchild=p->rchild ;
                else
                        f->rchild=p->rchild ;
                free(p);
}
else
{
        q=p;
        s=p->lchild;
        while(s->rchild)
        {
                q=s;
                s=s->rchild;
        }
        if(q==p)
                q->lchild=s->lchild ;
```

```
        else
                q->rchild=s->lchild;
        p->key=s->key;
        free(s);
}
return t;
```

(5) 执行及可视化

在主函数中完成功能实现并利用 Dot 和 Graphviz 工具对生成的 BST 进行可视化，代码略。

平衡二叉树的实现

(1) AVL 数据结构定义

平衡二叉树树的构建需要树型结构，因此 AVL 的数据结构需包含树结构的属性，包括值（key）、左孩子指针域（*lchild）和右孩子指针域（*rchild），根据 AVL 的特性，在插入的过程中需要一个平衡因子 bf 作为辅助标志变量。结构定义代码如下：

```
typedef int KeyType;

typedef struct  node
{
        KeyType  key ;
        int bf;
        struct node  *lchild,*rchild;
}AVLTNode, *AVLTree;
```

(2) AVL 构建流程

AVL 的构建与 BST 构建的总体流程类似，都是通过设置根节点初始化一棵树，然后不断向树中插入新数据，如下：

```
void  CreateAVLT(AVLTree  *bst, char * filename)
{
        FILE *fp;
        KeyType keynumber;
        *bst=NULL;
        fp = fopen(filename,"r+");
        while(EOF != fscanf(fp,"%d",&keynumber))
                ins_AVLtree(bst, keynumber);
}
```

(3) AVL 节点插入算法

向 AVL 插入节点的方法较为复杂。

根据实验原理，首先找到相应的插入位置，并记录离插入位置最近的可能失

衡节点 A，具体实现中，先查找待插入节点 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。之后，插入新节点 S。代码段如下：

```
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;
```

之后，确定节点 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->b
}
```

然后修改 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;
        }
```

接下来，需要根据 A、B 的平衡因子判断是否失衡以及失衡类型，根据失衡类型选择对应的方法进行相应处理。处理方式的具体选择如下：

- A 的平衡因子为 2，B 的平衡因子为 1：LL 型平衡旋转
- A 的平衡因子为 2，B 的平衡因子为-1：RR 型平衡旋转
- A 的平衡因子为-2，B 的平衡因子为 1：LR 型平衡旋转
- A 的平衡因子为-2，B 的平衡因子为-1：RL 型平衡旋转

每一种处理方式对应的代码段如下：

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;
```

RR 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;
```

LR 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;
```

RL 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;
```

(4) 执行创建 AVL 及可视化操作（代码略）

编译运行程序，之后在实验素材中运行“runme.bat”，生成结果。

七、实验数据及结果分析

二叉树的生成（BST 和 AVL）

在测试中，构造了 100 个数据元素序列，以-1 作为结束标记。测试序列如下：

247	323	808	622	966	331	336	491	729	266
473	528	16	606	752	663	588	144	542	401
568	271	320	972	835	244	219	708	845	351
41	989	25	691	998	691	961	433	165	422
259	82	257	117	475	340	28	428	389	75
185	763	766	262	107	10	339	0	467	967
433	621	46	290	636	702	244	58	116	700
358	861	118	228	399	783	739	826	802	185
994	970	82	497	201	693	106	79	955	651

983	23	57	608	513	624	650	63	612	611
-1									

表 1 测试序列 1

得到的排序二叉树如图二所示。

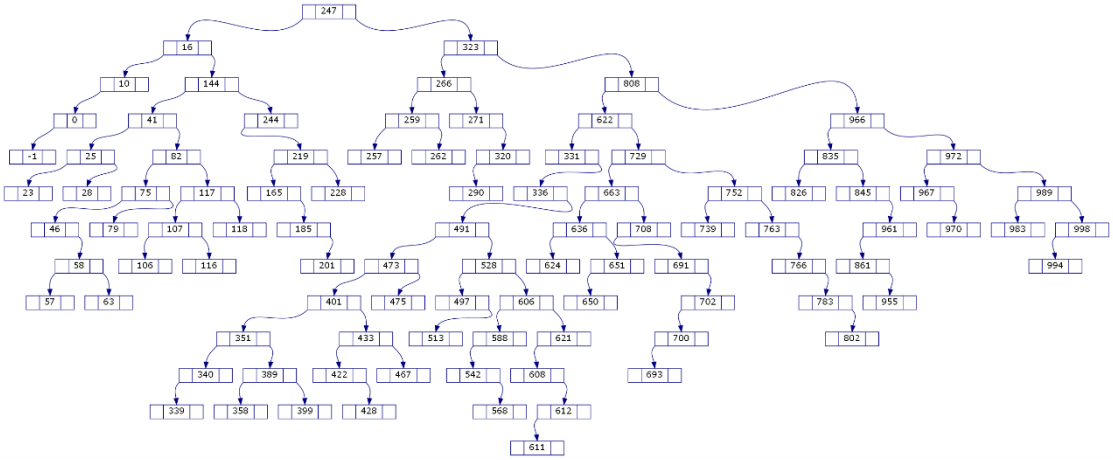


图 2 二叉排序树生成结果图（测试序列 1）

对于同样的输入序列，生成得到的平衡二叉树如图 2 所示。

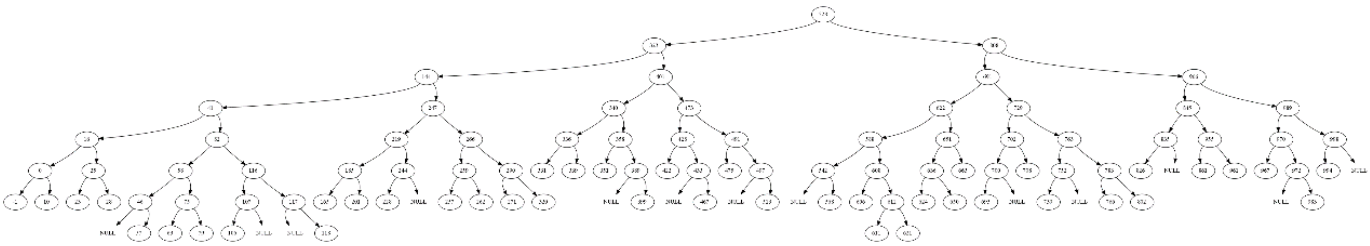


图 3 平衡二叉树生成结果图（测试序列 1）

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

为了进一步体现出平衡二叉树的优点，使用有序的数据作为输入，分别生成 BST 和 AVL。考虑到对有序数据生成 BST 深度过大，故仅构造含 20 个数据元素的序列，如下：

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
-1									

表 2 测试序列 2

得到的排序二叉树如图一所示。

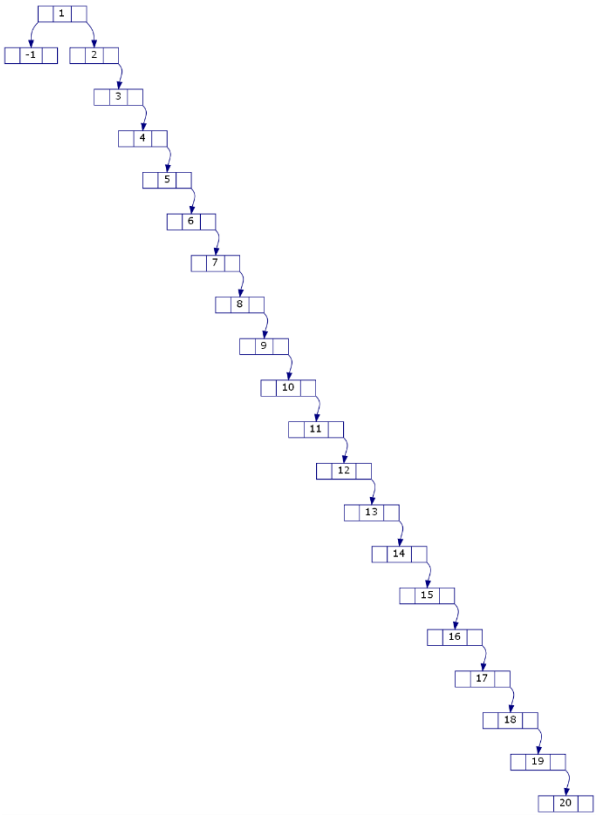


图 4 平衡二叉树生成结果图（测试序列 2）

对于同样的输入序列，生成得到的平衡二叉树如图 2 所示。

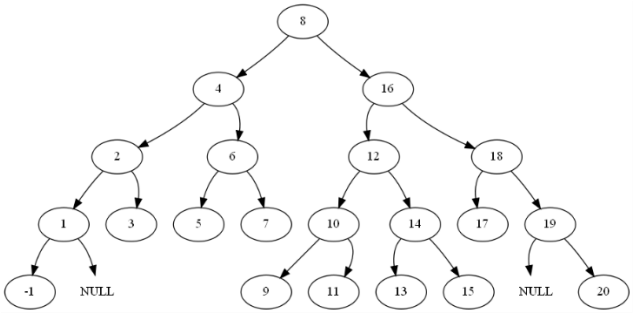


图 5 平衡二叉树生成结果图（测试序列 2）

可见，对于最坏情况，二叉排序树退化为链表，树的深度与数据元素的个数相同，而平衡二叉树则依然保持同节点数二叉树的最小深度。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。

BST 的按值查找

对于刚才的测试序列 1，进行 BST 的按值查找，分别查找 116 和 700。查找结果如下：

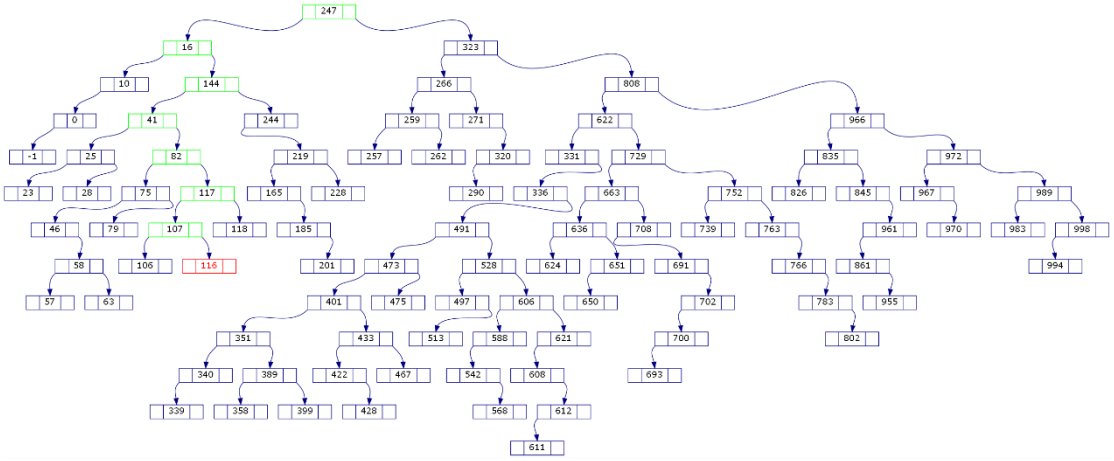


图 6 BST 的按值查找结果图 1（测试序列 1）

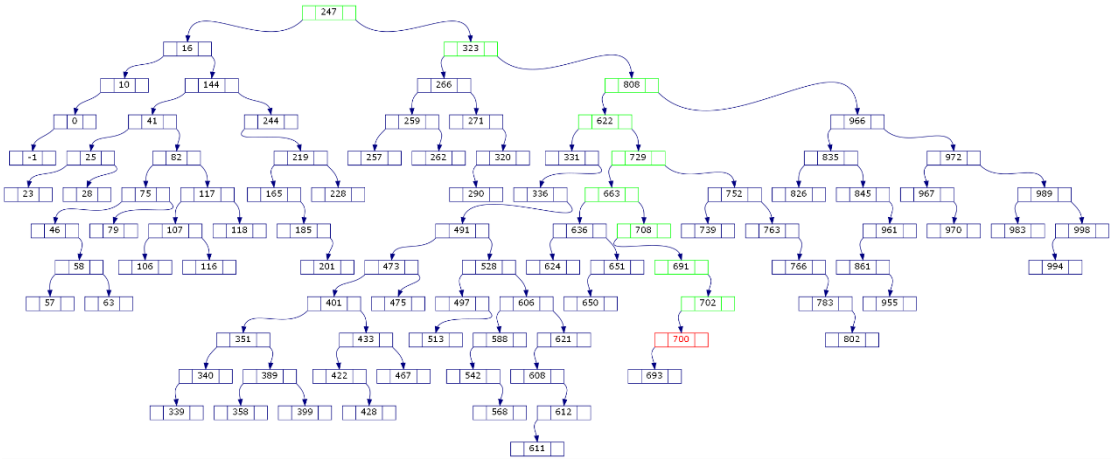


图 7 BST 的按值查找结果图 2（测试序列 1）

经分析，该查找路径与 BST 查找的实际查找路径相同，表明算法正常、正确运行，得到了预期结果。

BST 的按值删除节点

首先对刚刚所查找到的值为 116 的叶子节点进行删除，运行结果如图所示：

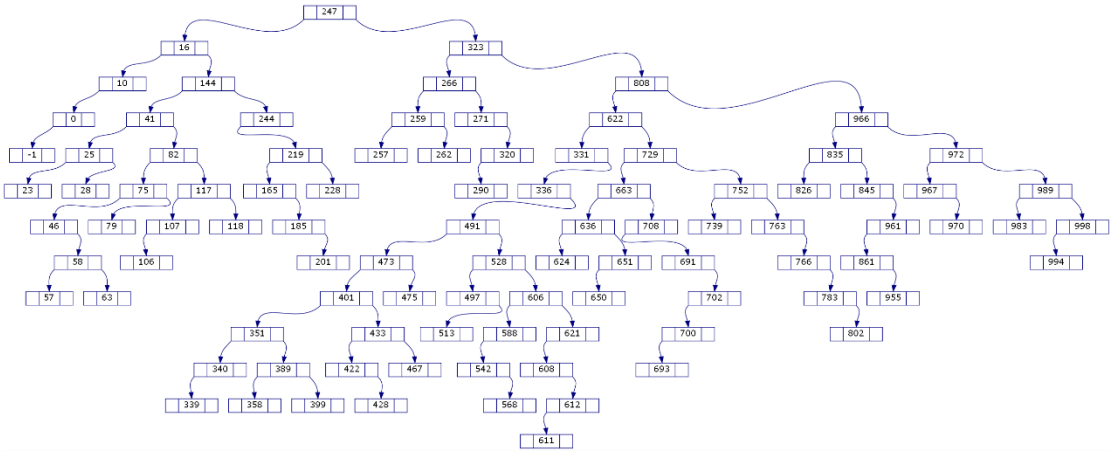


图 7 BST 的按值删除结果图 1（测试序列 1）

可见，该叶子节点成功从 BST 中删除。

接下来对度为 1 的节点进行删除。尝试删除刚刚查找到的值为 700 的节点，结果如下：

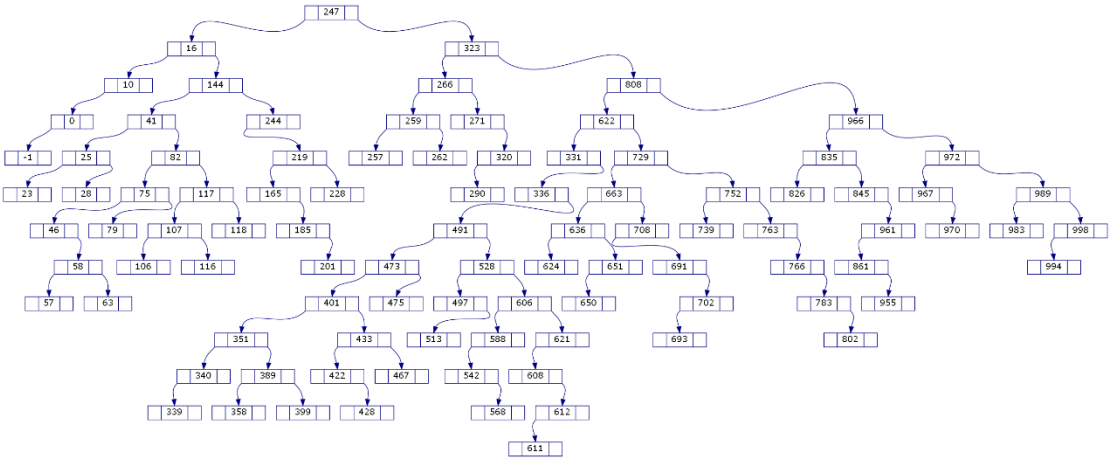


图 8 BST 的按值删除结果图 2（测试序列 1）

可见，该度为 1 的节点已成功删除，其父节点的子节点以 700 的非空子节点 693 代替之。结果符合预期。

再对度为 2 的节点进行删除。尝试删除 BST 第四层最右侧的节点 966，结果如下：

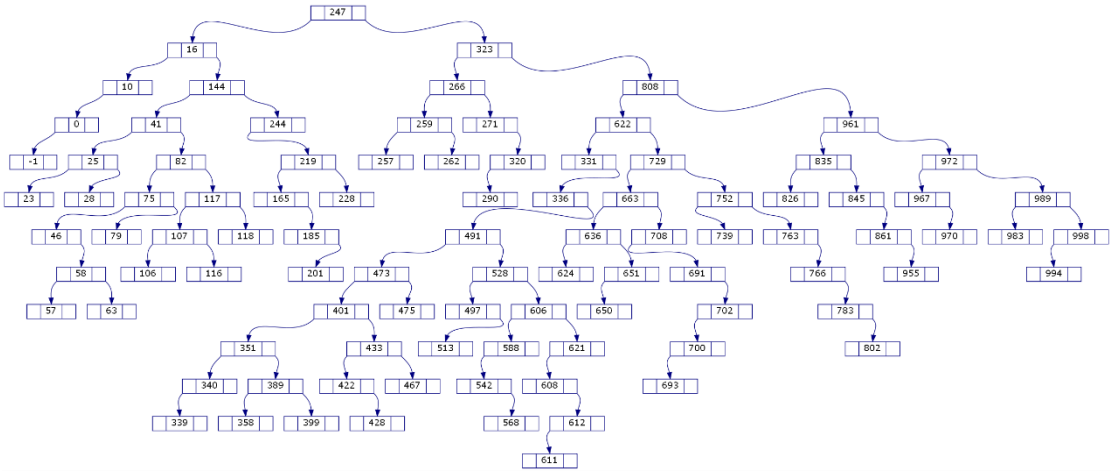


图 9 BST 的按值删除结果图 3（测试序列 1）

根据实验结果，该节点已被成功删除，其父节点的子节点以 966 的前驱 961 代替，而 961 是度为 1 的节点，因此删除 961 重新插入到 966 父节点的子节点中时，用 961 的非空子节点代替 961。到此删除节点带来的影响已经全部解决，得到了删除 966 之后的新 BST。实验结果符合预期。

八、总结及心得体会：

对随机序列构建了 BST 和 AVL，并完成了 BST 节点的按值查找和删除。通过对比相同测试序列构建的 BST 与 AVL，可以得到，生成的 BST 的深度具有较大的不确定性，而 AVL 经过调整，总能将树的深度控制在最小。引入二叉树来存储数据用于查找的优点源于折半法的思想,但BST并不是严格意义上的折半，参差不齐的BST将导致树的优点弱化,其时间复杂度介于 $O(\log_2 n)$和 $O(n)$之间。树的深度越大，对应的平均查找时间就越长。

实验让我更加深刻地掌握了 BST 和 AVL 的原理及应用，遗憾的是由于代码能力差、时间规划不合理，因此没能自己编码实现实验要求，失去了一个锻炼自己的宝贵机会，希望以后有机会朝花夕拾。

九、对本实验过程及方法、手段的改进建议：

感觉如果自己实现这些算法对代码能力要求好高,像我这样不熟悉编程语言、缺乏程序设计经验的学生觉得阻力很大。可以给出示例伪代码，既能指导实验，又能督促我们自己思考。

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

由于 Top-K 问题只需要找到 N 个数中最大的 K 个数，不需要知道其它数的排序情况，因此没有必要对所有的 N 个记录都进行排序。正常思路是维护一个 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，并对该数组进行排序，将数组中新的最小值放在最后一位，之后重复上述过程。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，因此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。

面对这样一种有排序思想、但不等同于排序的问题，很容易让人联想到堆积树，这种排序思想与 Top-K 问题的目标高度一致，并且比上述算法高效，因此利用堆进行优化。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）

处理器：	Intel® Core™ i5-7200U CPU @ 2.50GHz 2.70GHz
已安装的内存(RAM)：	4.00GB (3.81GB 可用)
系统类型：	64 位操作系统，基于 x64 的处理器

编程语言：	C++ 17
IDE：	Jetbrains CLion 2019.2.5
环境：	MinGW-W64 GCC-8.1.0
编译器：	g++ (i686-win32-dwarf-rev0) 8.1.0
调试器：	GNU gdb (GDB) 8.1
可视化：	Graphviz 2.39

六、实验步骤

编写堆调整函数用于堆积树的更新维护，利用函数传入的数组构建小顶堆，并重置父亲节点与儿子节点，通过不断交换找到子节点的最大值。实现该功能的伪代码如下：

1. BUILD-MAX-HEAP(*A*)
2. for *i* = *A.length* downto 2 do
3. exchange *A*[1] with *A*[i]
4. *A.heap-size* = *A.heap-size* – 1
5. MAX-HEAPIFY(*A*,1)
6. end for

使用 C++代码实现堆的调整 HeapAdjust 函数如下：

```
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;
    }
}
```

主程序从 Num.txt 文件中读取数据，初始化堆并反复调用 HeapAdjust 函数维护，通过 Dot 可视化工具绘制维护过程。代码略。

运行时间测试

固定 k=1024，更改 e 的值，分别取 e=15,20,25,27,29,30,31,32，测试运行时长并记录，根据时长绘制曲线。再固定 e=29，分别取 k=2^5、2^6、2^7、2^8、2^9、2^{10}、2^{11}、2^{12}，测试并记录运行时长，绘制曲线。

堆排序过程可视化

执行可视化操作（代码略）。编译运行程序，之后在实验素材中运行“runme.bat”，生成结果。

七、实验数据及结果分析

分别测试 k=1024，e=15,20,25,27,29,30,31,32 时的时长 t。选取 k=1024 的一列，以 e 作为横坐标，t 为纵坐标，绘制散点图 1 所示。可见，t 与 e 近似成指数关系。到 e=32 的时候，单次测试已经需要很长时间了。

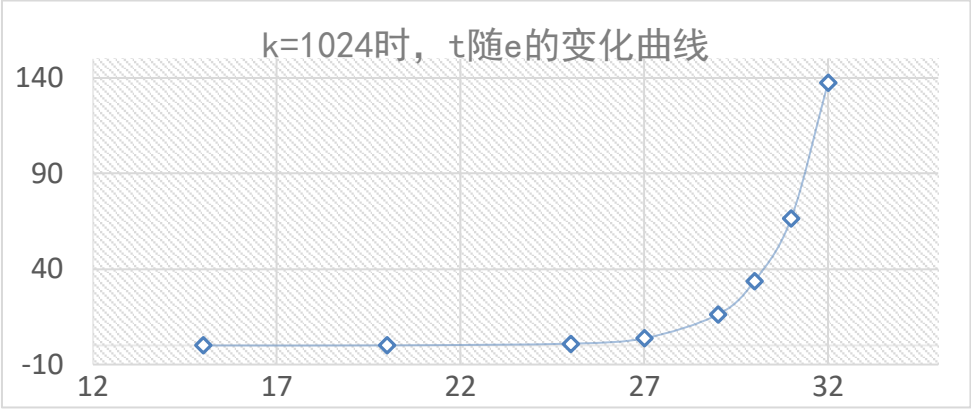


图 1 k=1024 时，t 随 e 的变化曲线

以 log2（t）为纵坐标，得到散点图如图 2 所示。可见，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。

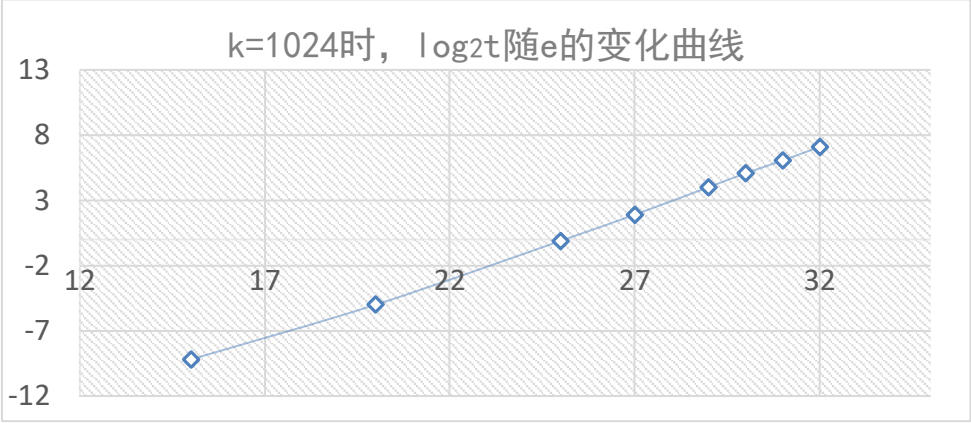


图 2 k=1024 时，log2t 随 e 的变化曲线

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，耗时与所构造的小顶堆的深度成正比。

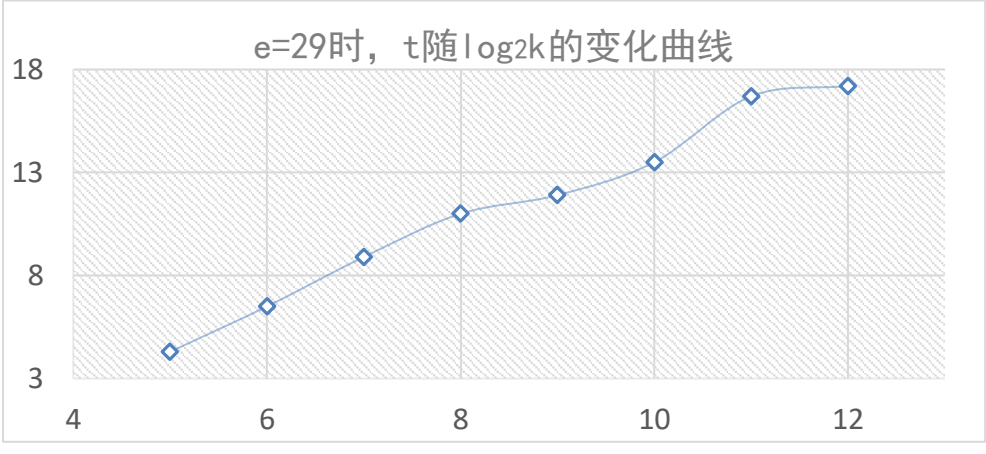


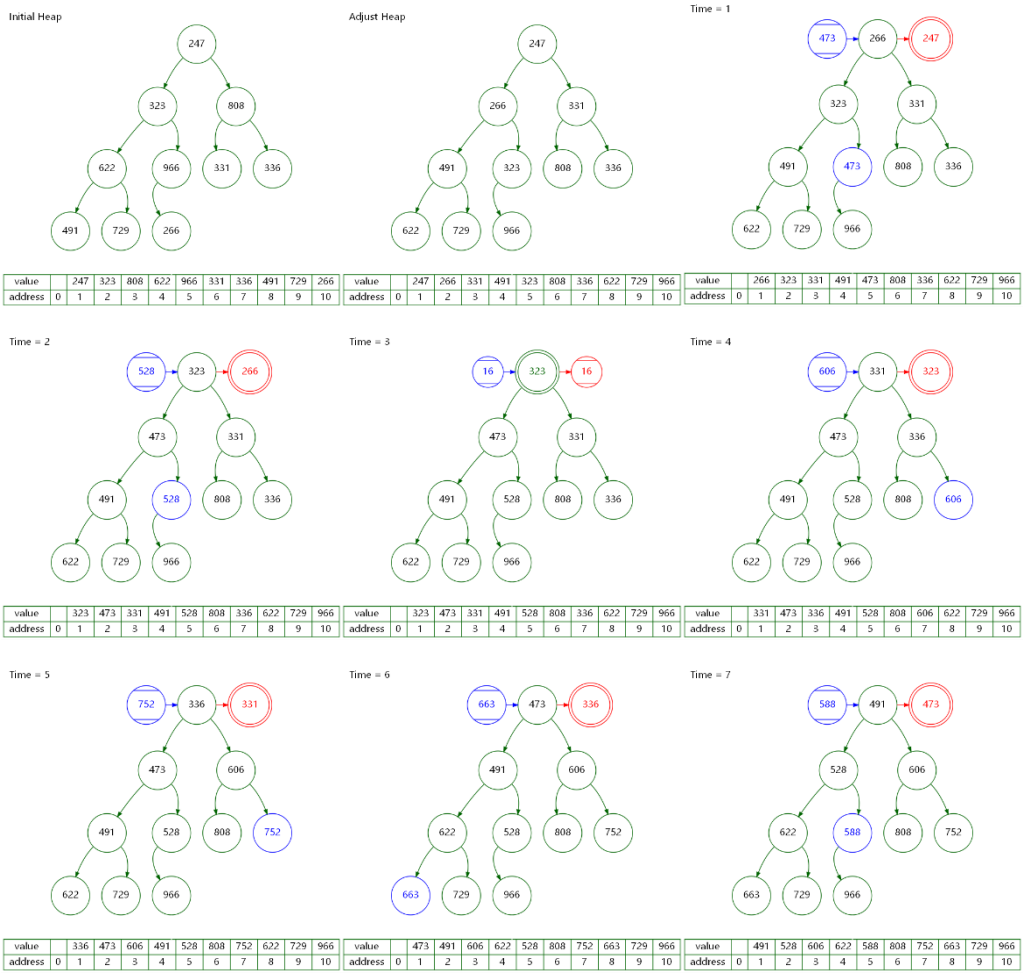
图 3　e=29 时，t 随 $\log_2 k$ 的变化曲线

在测试中，使用以下 30 个数据构成的元素序列进行测试。

247	323	808	622	966
331	336	491	729	266
473	528	16	606	752
663	588	144	542	401
568	271	320	972	835
244	219	708	845	351

表 1　测试序列

构造了堆调整的每个过程，如图 4 所示。



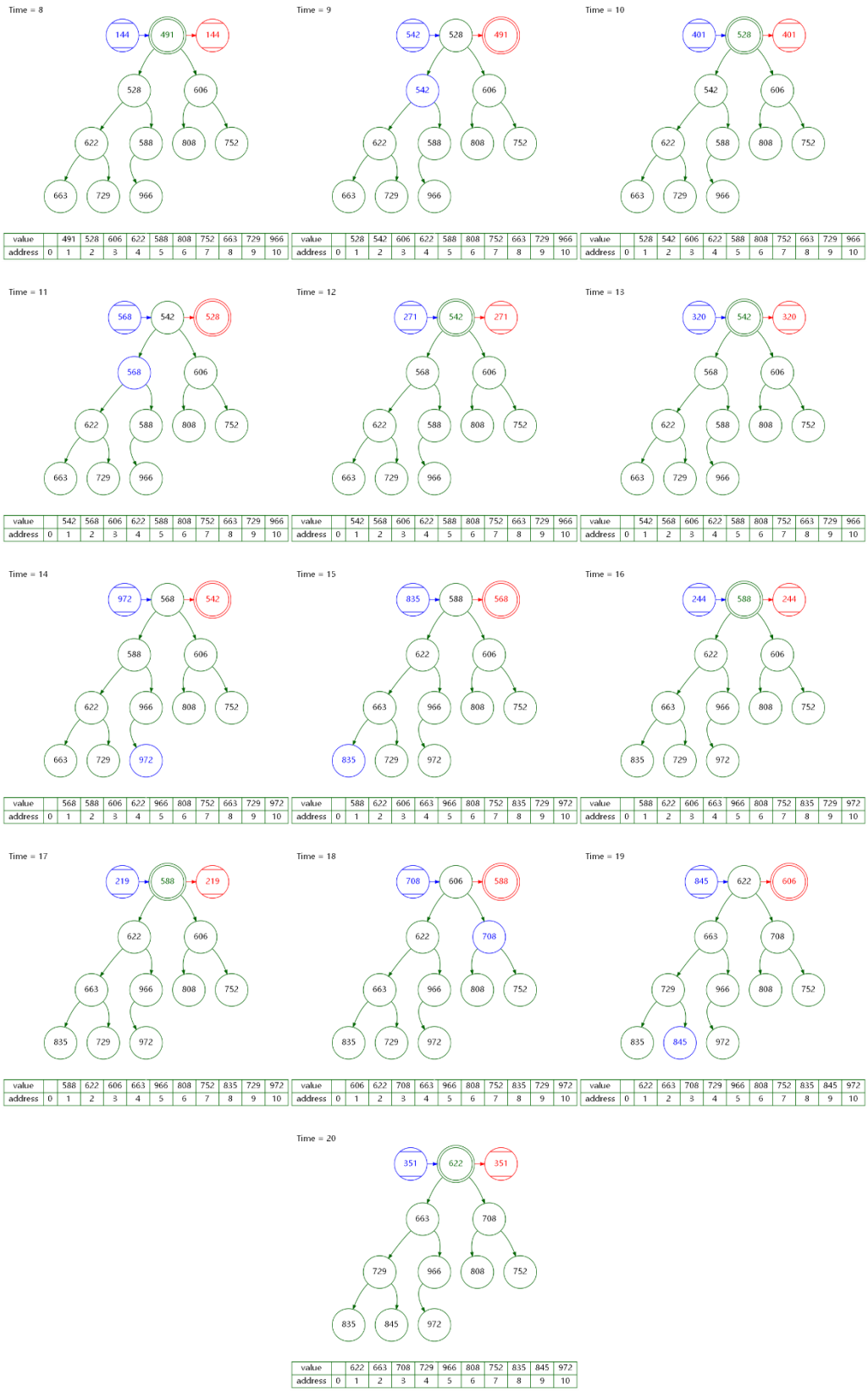


图 4　一次 Top10 的小顶堆调整过程

通过上述实验，经过堆操作，成功从 30 个数中找到了最大的 10 个数，即 622，663，708，729，966，808，752，835，845，972，完成了 Top-K 问题的一

个简单实现。

八、总结及心得体会：

Top-K 问题与堆积树，一个是日常生活中非常常见的问题，另一个是似乎仅仅漂浮在理论层面、在算法书中才会出现的不常用算法，两者看似毫无关系，但实际上后者却恰恰是前者的一个实用而贴切的解决方案。本实验利用堆的算法思想实现了 Top-K 问题的求解，让我了解到了堆积树的应用场景，并且加深了对堆排序的理解和认识。遗憾的地方在于由于时间紧张，没能自己编码实现 Top-K 问题的解决方案，使用了老师的参考代码，希望有机会可以再尝试。此外，测试的部分也有点偷工减料，比如 e=15~25 之间时采样间距是 5，后面变化大了才把间距改成 1，不过我觉得对于规律的验证点到为止也无可厚非。

九、对本实验过程及方法、手段的改进建议：

示例程序 142 行疑似“i=1”写成了“i=11”，输出图片时 Time 序号会有问题。

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。

五、实验器材（设备、元器件）

处理器：	Intel® Core™ i5-7200U CPU @ 2.50GHz 2.70GHz
已安装的内存(RAM)：	4.00GB (3.81GB 可用)
系统类型：	64 位操作系统，基于 x64 的处理器

编程语言：	C++ 17
IDE：	Jetbrains CLion 2019.2.5
环境：	MinGW-W64 GCC-8.1.0
编译器：	g++ (i686-win32-dwarf-rev0) 8.1.0
调试器：	GNU gdb (GDB) 8.1
可视化：	Graphviz 2.39

六、实验步骤

1. 定义数据结构

本实验实现有向图单源最短路径 Dijkstra 算法，故需要有向图数据结构，而对于代码实现而言，邻接矩阵是一种既直观、又便捷的存储方式。因此定义有向图数据结构，主要包括邻接矩阵**matrix，并定义顶点数 n 和边数 e。其中，matrix 矩阵的第 i 行第 j 列的元素表示从源点 i 到顶点 j 的直接边长度。结构定义代码如下：

```
typedef struct node
{
    int **matrix;
    int n;
    int e;
}MGraph;
```

2. 最短路径算法构建

由于 Dijkstra 算法的目的是确定图中某一点到其它所有节点的最短路径，因此函数的构造需要传入图本身 g 和指定节点的序号 vs。而在函数的处理过程中和函数完成处理之后，需要不断地更新当前最短路径和最短路径长度用于计算最短路径和返回结果。因此还需要传入数组指针*dist 和*path，分别用于存储最短路径下标和源点到个点最短路径的权值和。函数定义声明如下：

```
void DijkstraPath(MGraph g,int *dist,int *path,int vs);
```

3. 最短路径算法详细设计

在有向图 matrix 的定义中，我们定义无穷大常量 INT_MAX=65535，用来表示图中不直接相连的节点之间的距离无穷大。

对于有向图 Dijkstra 算法的实现，详细设计如下：

首先初始化 dist 和 path 数组，遍历除源点外的所有节点，得到每个节点与源点的初始距离，若直接相连则为弧的权值，并更新前驱为源点，否则为无穷大，前驱下标记为-1。与此同时，定义并初始化一个用于判断该节点是否已找到最短路径的数组 visited。并将源点自身的前驱设为 vs、聚源点距离设为 0。代码如下：

```
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;
```

至此，我们已找到了从源点（v0）访问 v0 的最短路径和最短距离，即 v0 访问自身，距离为 0。由于实验侧重对过程的可视化，因此即时输出可视化结果，将 v0 变成红色表明以确定为已找到最短路径节点。代码如下：

```
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);
```

并完整整个图中所有节点和弧的绘制。代码略。

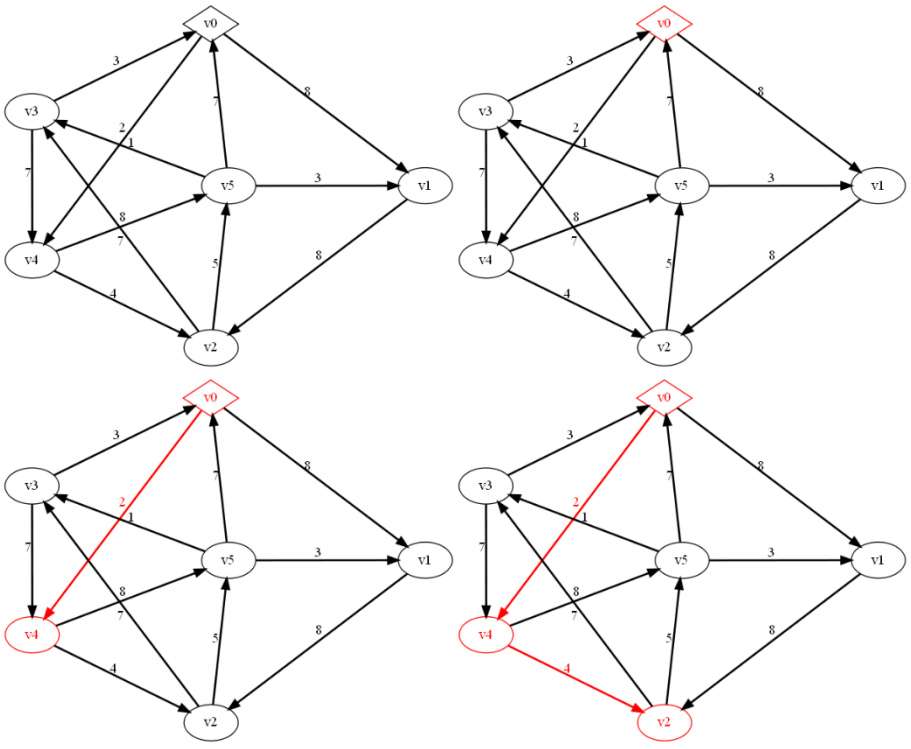
之后，利用贪心的思想，每次向前确定一个已找到最短路径上的节点，并通过此节点便利被邻接节点，确定下一个已找到最短路径节点。并将新拓展的路径写入 path，并比较 dist[k]+g[k,j]与 dist[j]的大小更新 dist。对于每一个新确定的节点，直到所有的节点都已经遍历完成，确定了从源点到最后一个节点的最短路径，而源点到任一节点的最短路径也能通过 path 中记录的前驱链接到最短路径中。代码如下：

```
visited[vs]=true;
for(i=1; i<g.n; i++)
{
```

```
    int min=INT_MAX;
    int u;
    for(j=0;j<g.n;j++)
    {
        if(visited[j]==false&&dist[j]<min)
        {
            min=dist[j];
            u=j;
        }
    }
    visited[u]=true;
    for(k=0;k<g.n;k++)
    {
        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;
        }
    }
}
```

七、实验数据及结果分析

通过程序及可视化工具生成的确定最短路径的过程如图 1 所示。



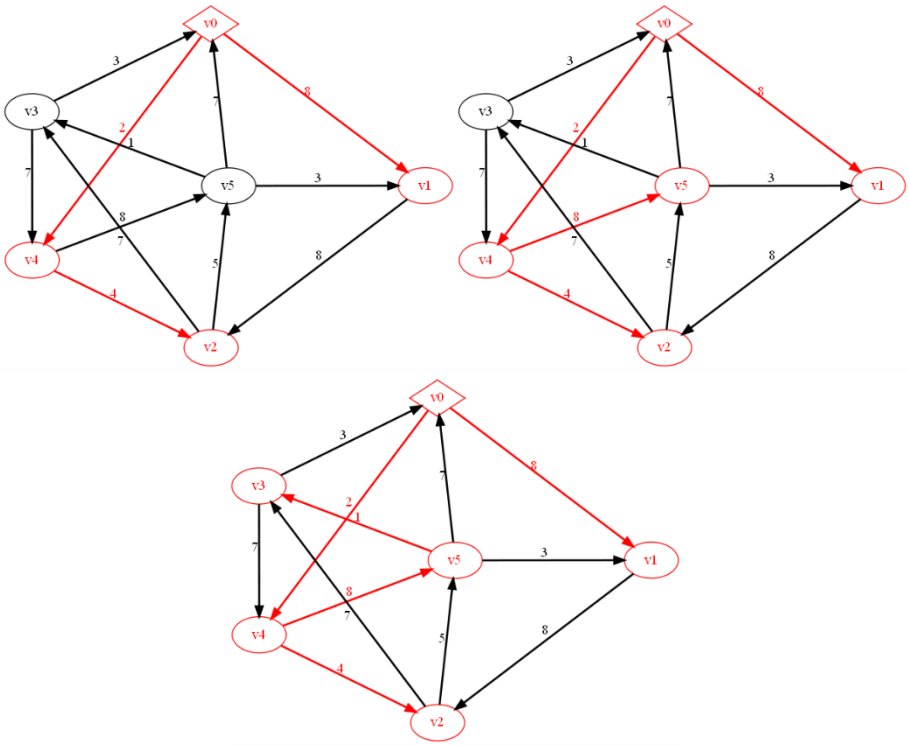


图 1　一次 6 个节点 12 条边有向图最短路径算法过程

通过结果，可以得到程序的执行逻辑：

(用 S 表示已计算出最短路径的顶点的集合，U 表示未计算出最短路径的顶点的集合)

1) 从 v0 出发，首先将 v0 确定为最短路径的第一个节点；

   S{v0(0)}, U{v1(8),v2(∞),v3(∞),v4(2),v5(∞)}

2) v1、v4 与 v0 邻接，比较 v0 到两个节点的距离，发现 v4 更近，将 v4 确定为下一个节点，同时更新 v5 到源点的距离；

   S{v0(0) ,v4(2)}, U{v1(8),v2(6),v3(∞),v5(10)}

3) v2、v5 与 v4 邻接，比较 v4 到两个节点的距离，发现 v2 更近，将 v2 确定为下一个节点，同时更新 v3 到源点的距离；

   S{v0(0) ,v4(2) ,v2(6)}, U{v1(8),v3(14),v5(10)}

4) 确定 v1 的前驱为 v0。

   S{v0(0) ,v4(2) ,v2(6) ,v1(8)}, U{ v3(14),v5(10)}

5) v5 与 v2 邻接，但经 v2 访问 v5 的距离并不比 v5 的当前最短距离更短，因此 v5 的最短路径前驱为 v4，同时更新 v3 到源点的距离。

   S{v0(0) ,v4(2) ,v2(6) , v1(8),v5(10)}, U{v3(11) }

6) v3 的最短路径前驱确定为 v5。

   S{v0(0) ,v4(2) ,v2(6) , v1(8),v5(10) ,v3(11)}, U{}

7) 完成。

最终，得到从 v0 到任一节点的最短路径：

v0：v0

v1：v0->v1

v2：v0->v4->v2

v3：v0->v4->v5->v3

v4：v0->v4

v5：v0->v4->v5

八、总结及心得体会：

通过实验实现了 6 节点 12 边的有向图单源最短路径算法，并通过可视化工具对算法细节过程进行了逐步的图解。Dijkstra 算法能在错综复杂的图中找到单源最短路径，其中的很多思想和结论想想还是非常神奇的。尽管时间复杂度为 $O(n^2)$，但这丝毫不影响该算法对于解决最短路径问题的里程碑式的意义。

Dijkstra 算法从源点出发，一点一点向前拓展，直到为每一个节点找到最短路径的解决方案。结合可视化的图解，让我对该算法有了更清晰的认识和更深刻的掌握，尤其是学习和探索了其算法实现之后，接触了从逻辑可行到动手实践的跨越，也让我更加意识到自己在程序设计能力上的不足。

九、对本实验过程及方法、手段的改进建议：

可视化的时候可以在节点旁标注出当下该节点距源点的总距离，即 dist 数组中的内容，以便从图中得到更清晰、细致的信息，进而对最短路径的求解过程有更全面、深入的了解。

三个实验都用的示例代码。作为大三刚转专业的同学，一学期上两学期的课，压力大、时间紧，而且代码能力有待提升，希望老师理解。