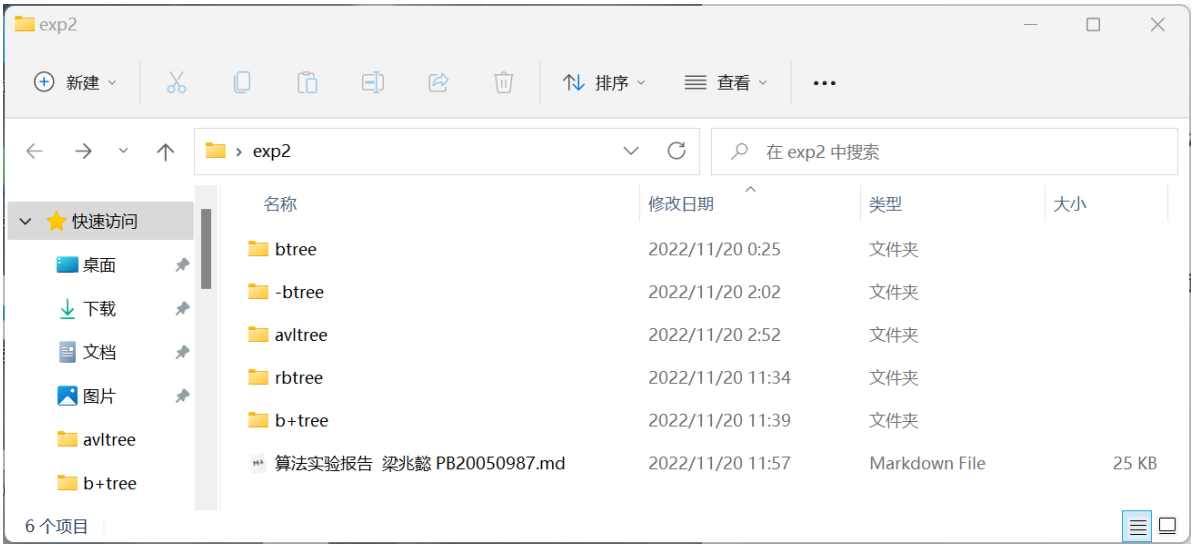
算法分析与设计

第二次实验报告

PB20050987梁兆懿

本次实验的文件打包在实验二的文件夹中。各树的实现方式分开存放，从上到下依次是二叉查找树，-B树，AVL树，红黑树，B+树。



(1) 实验内容

给定一个包含n个元素的实数数组，请构建一颗二叉搜索树，并实现节点的插入、删除和搜索过程；

实验记录：

1.使用大的随机数组进行验证。

随机数组生成函数代码如下：

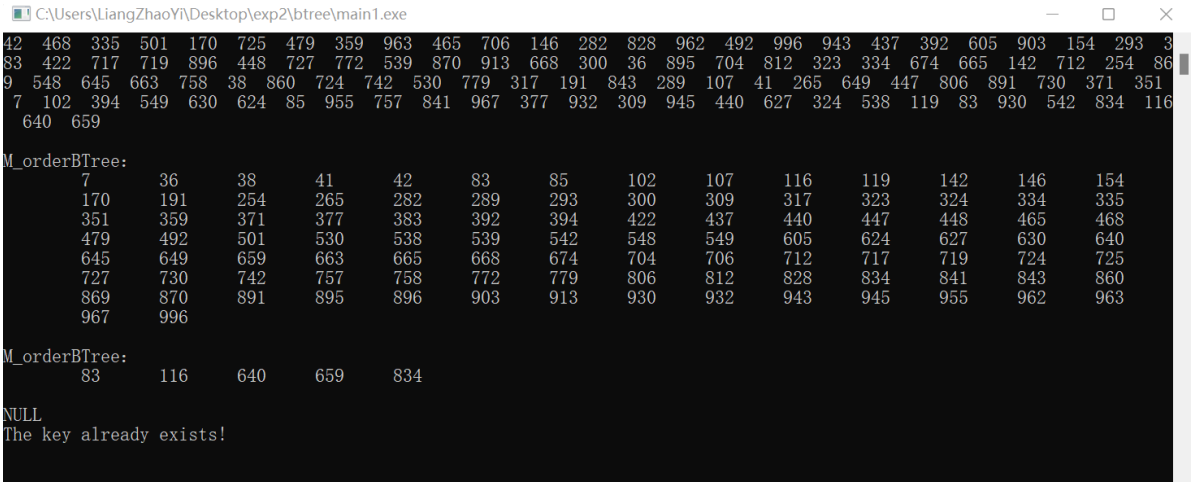
```
int num_rand(int a[],int size,int randsize)
{
    int i;
    int b[randsize+1];
    for(i=0;i<=randsize;i++){
        b[i]=0;
    }
    for(i=0;i<size;i++)
    {
        while(1){
        a[i]=rand()%randsize+1;
        if(b[a[i]]!=1){
            b[a[i]]=1;
            break;
        }
    }
}
}
```

可以生成大小为 size ， 范围为 randsize 的数组。

首先，我们用大小为100，范围为0~1000的数组进行简单验证。使用打乱的随机数组创建树，再中序遍历树，删除其中1~95个数，验证算法的正确性。

随后，我们查找数组中的30和83，验证搜索功能的正确性。

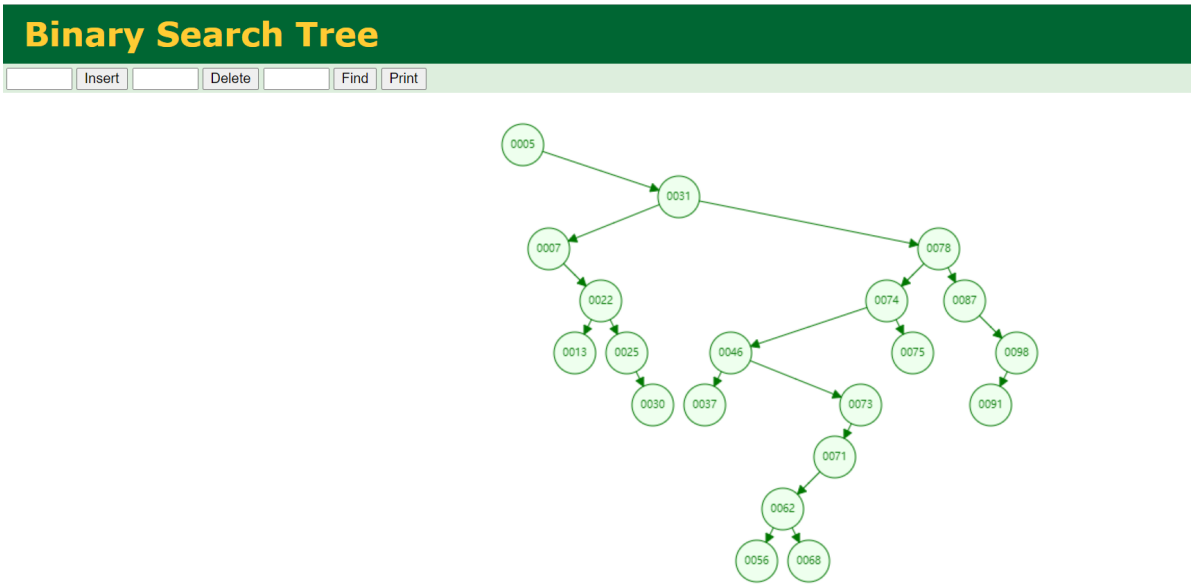
如下图所示。



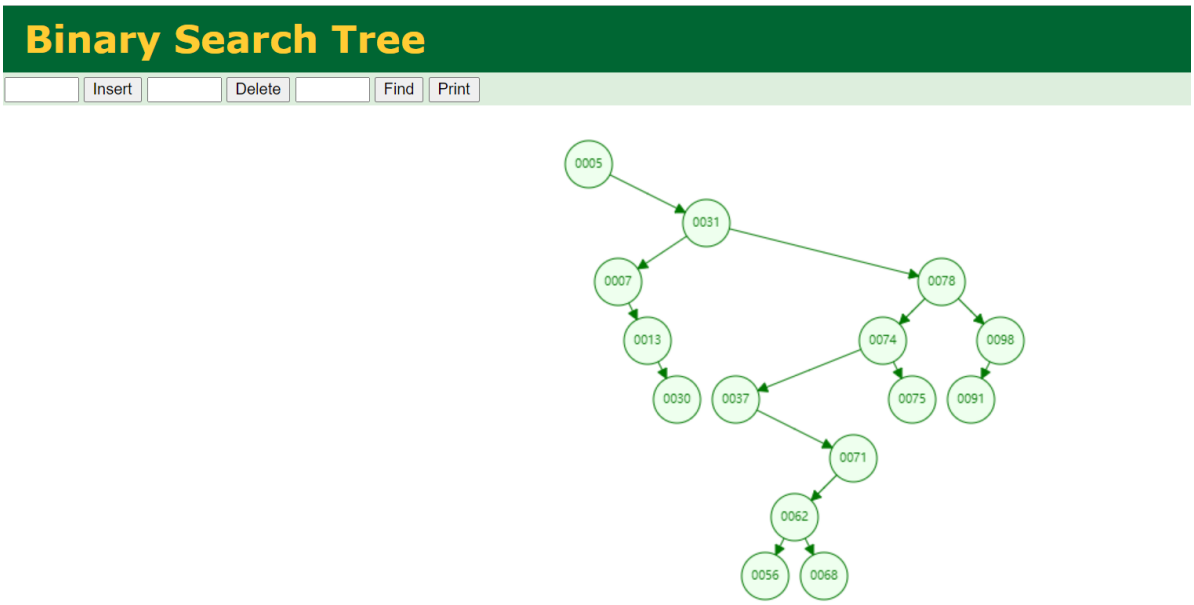
2.使用小随机数组验证树的结构。

我们再用范围为1~100，大小为20的随机数组验证数的结构是正确的。生成树并且打印：

再与助教给出网站中树的结构对比：



删除中间5个数后，得到如下数：



可见，树的结构是正确的。然而在删除后，我实现的二叉搜索树和网站上的树结构有出入，但是结构是正确的，仍然属于二叉搜索树。推测原因是在删除节点时，二者删除的方式不一样。在我的节点删除代码中，若节点的左子树存在，使用左子树的最右节点替代该节点。而在网站中实现方式可能略有不同。

实现删除的函数如下：

```
btree* DelBTree(btree* Btree, int X)
{
    btree* Tmp;
    if (!Btree)
    {
        printf("Not Found\n");
    }
    else if (X < Btree->data)
    {
        Btree->lchild = DelBTree(Btree->lchild, X);
    }
    else if (X > Btree->data)
    {
        Btree->rchild = DelBTree(Btree->rchild, X);
```

```
    }
    else
    {
        if (Btree->lchild && Btree->rchild)
        {
            Tmp = FindMin(Btree->rchild);
            Btree->data = Tmp->data;
            Btree->rchild = DelBTree(Btree->rchild, Btree->data);
        }
        else
        {
            Tmp = Btree;
            if (!Btree->lchild)
            {
                Btree = Btree->rchild;
            }
            else if (!Btree->rchild)
            {
                Btree = Btree->lchild;
            }
        }
    }
    return Btree;
}

//寻找结点
int FindOBTree(btree* root,int fnum)
{
    if (root == NULL)
    {
        return 0;
    }
    if (fnum < root->data)
    {
        if(FindOBTree(root->lchild, fnum)==1)
        return 1;
        else return 0;
    }
    else if (fnum > root->data)
    {
        if(FindOBTree(root->rchild, fnum)==1){
            return 1;
        }
        else return 0;
    }
    else {
        return 1;
    }
}
```

主函数实现代码如下：

```
int main()

{   int a[100] ;
    num_rand(a,100,1000);                              //使用大数组验证树的插入以及排序
    int i;
```

```
    for(i=0;i<100;i++){
        printf("%d  ",a[i] );
    }
    printf("\n\n");
    btree* root = (btree*)malloc(sizeof(btree));
    root = NULL;
    for(i=0;i<100;i++){
            btree* node = InitnodeBTree(a[i]);
            root = InsertBTree(root, node);
        }
    printf("M_orderBTree: \n\t");
    M_orderBTree(root);
    printf("\n\n");
    for(i=0;i<=95;i++)
        DelBTree(root, a[i]);
    printf("M_orderBTree: \n\t");
    M_orderBTree(root);
    printf("\n\n");

    FindBTree(root,30);
    FindBTree(root,83);
    printf("\n\n");                                    //验证搜索功能

    int b[20] ;
    num_rand(b,20,100);
    printf("\nrand list:\n");
    for(i=0;i<20;i++)
        printf("  %d  ",b[i] );
    btree* root1 = (btree*)malloc(sizeof(btree));
    root1 = NULL;
    for(i=0;i<20;i++){
            btree* node = InitnodeBTree(b[i]);
            root1 = InsertBTree(root1, node);
        }
        printf("\n\n");
    printf("M_orderBTree: \n\t");
    M_orderBTree(root1);
    printf("\n_____________________________________________________________________\n");

    PrintBTree(root1);
    printf("\n_____________________________________________________________________\n");
    for(i=5;i<10;i++)
        DelBTree(root1,b[i]) ;
    printf("after delete:\n");
    PrintBTree(root1);
    printf("\n_____________________________________________________________________\n");
                                                        //使用小数组验证树的结构
    int c[20000] ;                                      //测量树实现各功能需要的时间
    num_rand(c,20000,100000);

    printf("\n\n");
    btree* root2 = (btree*)malloc(sizeof(btree));
    root = NULL;

    for(i=0;i<1000;i++){
            btree* node = InitnodeBTree(c[i]);
            root2 = InsertBTree(root2, node);
        }
```

```
    clock_t begin = 0, end = 0;
    begin = clock();
    for(i=1000;i<20000;i++){
            btree* node = InitnodeBTree(c[i]);
            root2 = InsertBTree(root2, node);
        }
    end = clock();
    printf("\nInsert time: %5.3f\n",(float)(end-begin)/CLOCKS_PER_SEC);

    begin = clock();
    for(i=1000;i<20000;i++)
        DelBTree(root2,c[i]) ;
    end = clock();
    printf("\nDelete time: %5.3f\n",(float)(end-begin)/CLOCKS_PER_SEC);

    for(i=1000;i<20000;i++)
        btree* node = InitnodeBTree(c[i]);
        root2 = InsertBTree(root2, node);                //把删除的节点重新添加

    begin = clock();
    for(i=1000;i<20000;i++)
            FindOBTree(root2,c[i]);
    end = clock();
    printf("\nSearch time: %5.3f\n",(float)(end-begin)/CLOCKS_PER_SEC);
    return 0;
    }
```

(2) 实验内容

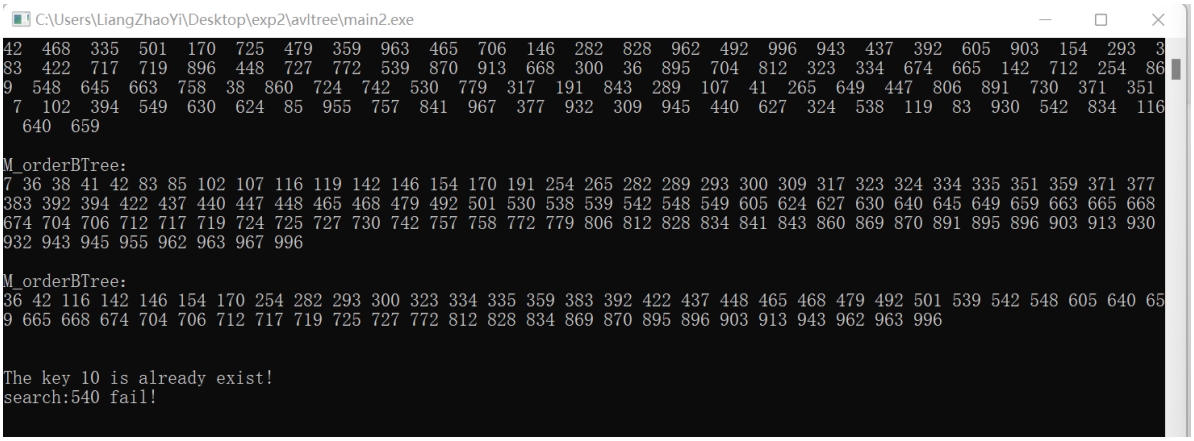
给定一个包含n个元素的实数数组，请构建一颗AVL树，并实现节点的插入、删除和搜索过程；

AVL树与平衡二叉树的主要区别是在插入和删除时需要加入旋转操作，保证二叉树是平衡二叉树。

实验记录：

1.使用大的随机数组进行验证。

首先，生成一个大小为100，范围为0~1000的数组，插入节点并形成树。随后删除其中50~95的数，中序遍历删除后的树。最后验证查找算法的正确性。



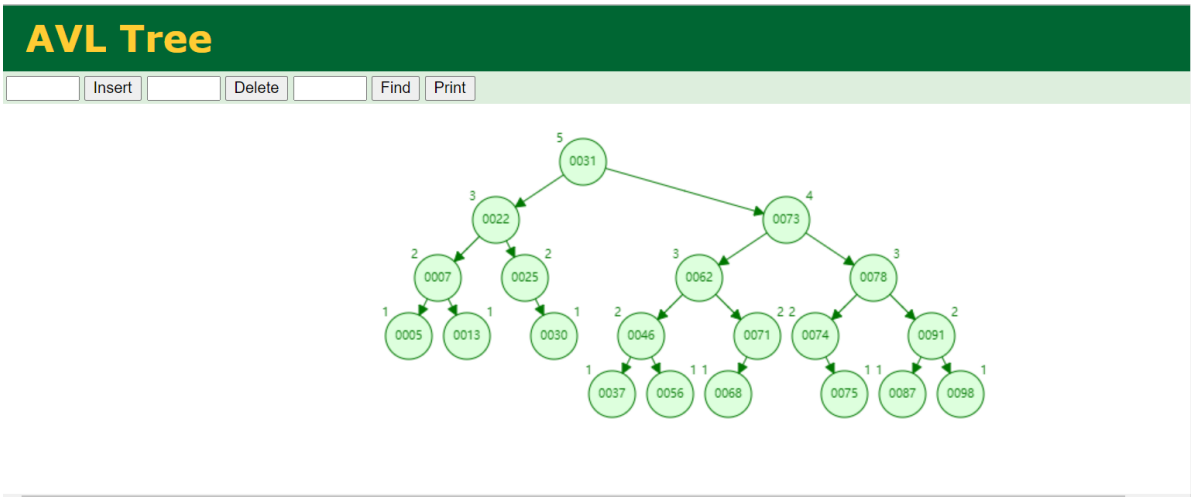
如图所示，可以正确形成树并且删除后树的结构没有被破坏。

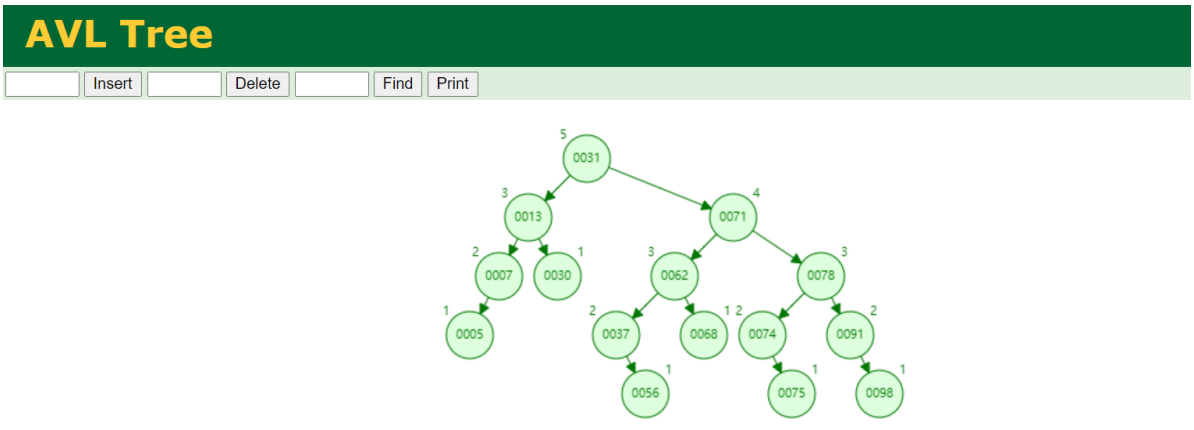
2.使用小规模的随机数组检验树的结构。

我们使用范围为1~100，大小为20的随机数组验证数的结构是正确的。生成树并且打印，如图所示：

```
C:\Users\LiangZhaoYi\Desktop\exp2\avltree\main2.exe
rand list:
  5    31    78    7    74    87    22    46    25    73    71    30    98    13    91    62    37    56    68    75

M_orderBTree:
       5 7 13 22 25 30 31 37 46 56 62 68 71 73 74 75 78 87 91 98
________________________________________________________________
          98
       91
          87
    78
          75
       74
  73
       71
          68
    62
          56
       46
          37
31
       30
    25
  22
       13
     7
       5


________________________________________________________________
after delete:
          98
       91
    78
          75
  74
       71
          68
    62
          56
       37
31
    30
  13
     7
       5
```

在网站上实现树的图形化，结果如图所示：





可以看出，结果仍是在删除操作后，树的结构出现差异。我在实现树的删除结构时，先使用二叉查找树节点删除方法，再通过旋转实现树的平衡。由于在实验一中二叉查找树删除的产生树的结构已经有所不同，而我沿用了其中删除的方法，因此难免导致在实验二中删除结果也有所出入，但是在平衡选择操作后仍是一棵正确的AVL树。

主函数实现代码如下：

```
int main()

{   int a[100] ;                                          //大规模数组检查树的结构
    num_rand(a,100,1000);
    int i;
    for(i=0;i<100;i++){
        printf("%d  ",a[i] );
    }
    printf("\n\n");
    AVLTreeN root = NULL;

    for (i =0; i<100; i++) {
        root = InsertAVLTree(root, a[i]);
    }

    printf("M_orderBTree: \n");
    InorderAVLTree(root);
    printf("\n\n");

    for(i=50;i<95;i++)
        root=DeleteAVLTree(root, a[i]);
    printf("M_orderBTree: \n");
    InorderAVLTree(root);
    printf("\n\n");

    if(SearchAVLTree(root, 42)!=NULL){
        printf("\nThe key %d is already exist!",10);
    }else
    printf("\nsearch fail!");
    if(SearchAVLTree(root, 540)!=NULL){
        printf("\nThe key is already exist!");
    }else
    printf("\nsearch:%d fail!",540);
    printf("\n\n");

    int b[20] ;                                           //检验树结构的正确性
    num_rand(b,20,100);
    printf("\nrand list:\n");
    for(i=0;i<20;i++)
        printf("  %d  ",b[i] );
    AVLTreeN root1 = NULL;

    for (i = 0; i < 20; i++) {
        root1 = InsertAVLTree(root1, b[i]);
    }
    printf("\n\n");
    printf("M_orderBTree: \n\t");
    InorderAVLTree(root1);
    printf("\n_____________________________________________________________\n");
```

```
    PrintAVLTree(root1);
    printf("\n_____________________________________________________________________\n");
    for(i=5;i<10;i++)
        DeleteAVLTree(root1,b[i]) ;
    printf("after delete:\n");
    PrintAVLTree(root1);
    printf("\n_____________________________________________________________________\n");

    int c[20000] ;
    num_rand(c,20000,100000);

    printf("\n\n");
    AVLTreeN root2 = NULL;

    for (i =0; i<1000; i++) {                          //测量相关功能实现时间
        root2 = InsertAVLTree(root2, c[i]);
    }
    clock_t begin = 0, end = 0;
    begin = clock();
    for(i=1000;i<20000;i++)
        root2 = InsertAVLTree(root2, c[i]);
    end = clock();
    printf("\nInsert time: %5.3f\n",(float)(end-begin)/CLOCKS_PER_SEC);

    begin = clock();
    for(i=1000;i<20000;i++)
        DeleteAVLTree(root2, c[i]);
    end = clock();
    printf("\nDelete time: %5.3f\n",(float)(end-begin)/CLOCKS_PER_SEC);

    for(i=1000;i<20000;i++)
        root2 = InsertAVLTree(root2, c[i]);

    begin = clock();
    for(i=1000;i<20000;i++)
        SearchAVLTree(root2, c[i]);
    end = clock();
    printf("\nSearch time: %5.3f\n",(float)(end-begin)/CLOCKS_PER_SEC);
    return 0;
}
```

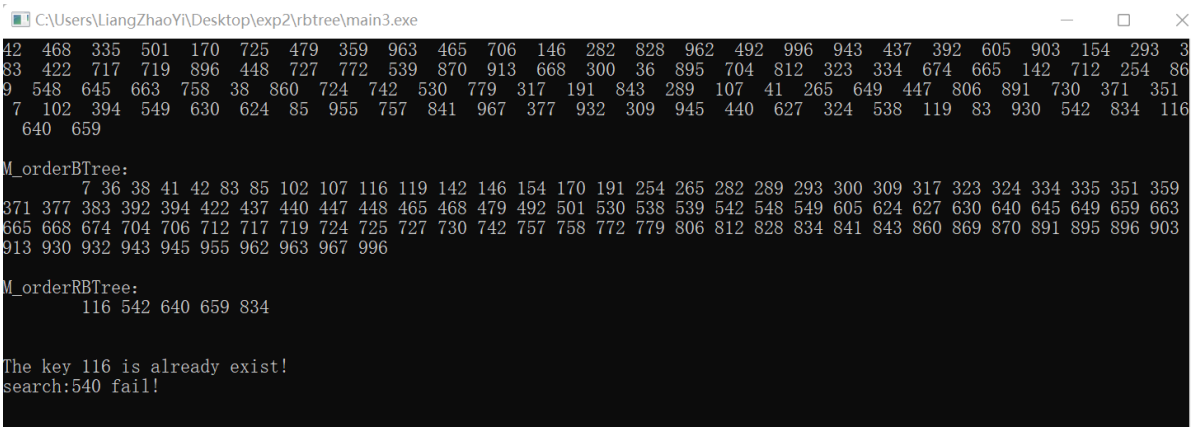
(3) 实验内容

给定一个包含n个元素的实数数组，请构建一颗红黑树，并实现节点的插入、删除和搜索过程；

红黑树树使用黑节点的个数代替深度，避免了过多的旋转操作。

1.使用大的随机数组进行验证。

首先，生成一个大小为100，范围为0~1000的数组，插入节点并形成树。随后删除其中0~95的数，中序遍历删除后的树。最后验证查找算法的正确性。

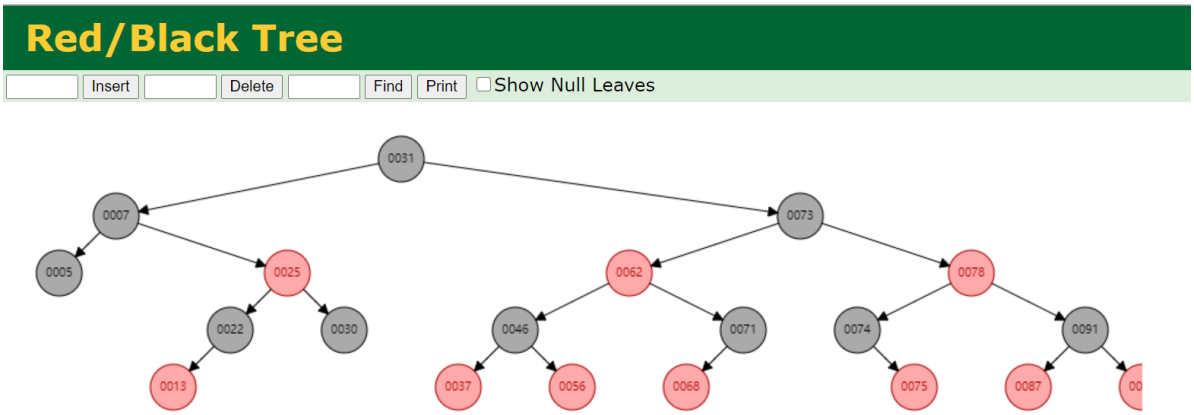


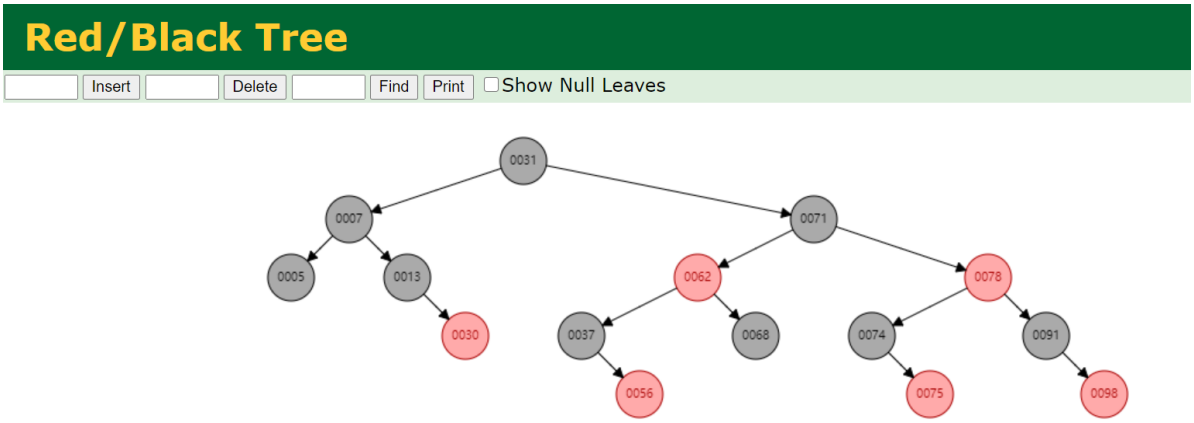
如图所示，可以正确形成树并且删除后树的结构没有被破坏。查找功能也能正确实现。

2.使用小规模的随机数组检验树的结构。

我们使用范围为1~100，大小为20的随机数组验证数的结构是正确的。生成树并且打印，如图所示：

再与网站上生成的红黑树进行对比，如图所示：





如图，在删除操作后，两棵红黑树的结构依然都是正确的，但红黑树的结构依然不同（悲）。可谓是一步错，步步错了。原因是在删除节点以及选择操作上的不同，导致了树最终结构不同。

主函数代码：

```
int main()
{   int a[100] ;                                                    //大规模数组验证
    num_rand(a,100,1000);
    int i;
    for(i=0;i<100;i++){
        printf("%d  ",a[i] );
    }
    printf("\n\n");
    RBRoot *root=NULL;
    root = CreateRBTree();

    for (i =0; i<100; i++) {
        InsertRBTree(root, a[i]);
    }

    printf("M_orderBTree: \n\t");
    InorderRBTree(root);
    printf("\n\n");

    for(i=0;i<95;i++)
        DeleteRBTree(root, a[i]);
    printf("M_orderRBTree: \n\t");
    InorderRBTree(root);
    printf("\n\n");
```

```
    if(SearchRBTree(root, 116)!=-1){
        printf("\nThe key %d is already exist!",116);
    }else
    printf("\nsearch 116 fail!");
    if(SearchRBTree(root, 540)!=-1){
        printf("\nThe key is already exist!");
    }else
    printf("\nsearch:%d fail!",540);
    printf("\n\n");

    int b[20] ;                                          //小规模数组验证树的结构
    num_rand(b,20,100);
    printf("\nrand list:\n");
    for(i=0;i<20;i++)
        printf("  %d  ",b[i] );

    RBRoot *root1=NULL;
    root1 = CreateRBTree();
    for (i = 0; i < 20; i++) {
        InsertRBTree(root1, b[i]);
    }
    printf("\n\n");
    printf("M_orderBTree: \n\t");
    InorderRBTree(root1);
    printf("\n_____________________________________________________________________\n");

    Print1RBTree(root1);
    printf("\n_____________________________________________________________________\n");
    for(i=5;i<10;i++)
        DeleteRBTree(root1,b[i]) ;
    printf("after delete:\n");
    Print1RBTree(root1);
    printf("\n_____________________________________________________________________\n");

    int c[20000] ;                                       //测量树实现有关功能的时间
    num_rand(c,20000,100000);

    printf("\n\n");
    RBRoot *root2=NULL;
    root2 = CreateRBTree();
    for (i = 0; i < 1000; i++) {
        InsertRBTree(root2, c[i]);
    }

    clock_t begin = 0, end = 0;
    begin = clock();
    for(i=1000;i<20000;i++)
        InsertRBTree(root2, c[i]);
    end = clock();
    printf("\nInsert time: %5.3f\n",(float)(end-begin)/CLOCKS_PER_SEC);

    begin = clock();
    for(i=1000;i<20000;i++)
        DeleteRBTree(root2, c[i]);
    end = clock();
    printf("\nDelete time: %5.3f\n",(float)(end-begin)/CLOCKS_PER_SEC);

    for(i=1000;i<20000;i++)
```

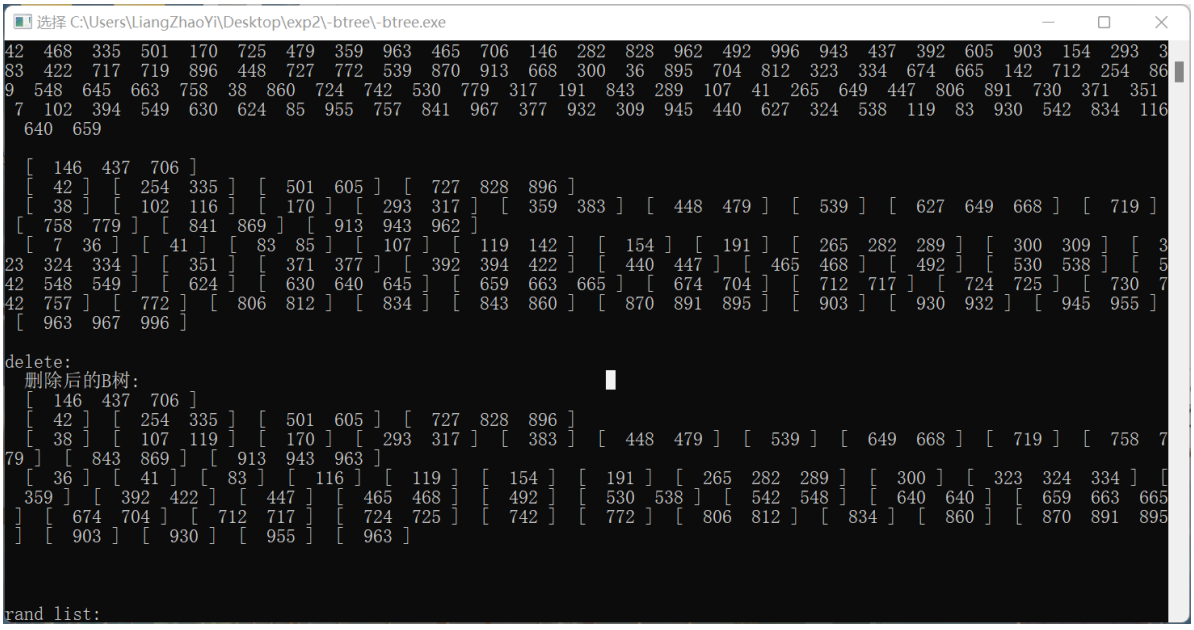
```
        InsertRBTree(root2, c[i]);

    begin = clock();
    for(i=1000;i<20000;i++)
        SearchRBTree(root2, c[i]);
    end = clock();
    printf("\nSearch time: %5.3f\n",(float)(end-begin)/CLOCKS_PER_SEC);
    return 0;
}
```

(4) 实验内容

给定一个包含n个元素 的实数数组，请构建一颗-B树，并实现节点的插入、删除和搜索过程；

1.使用大的随机数组进行验证。

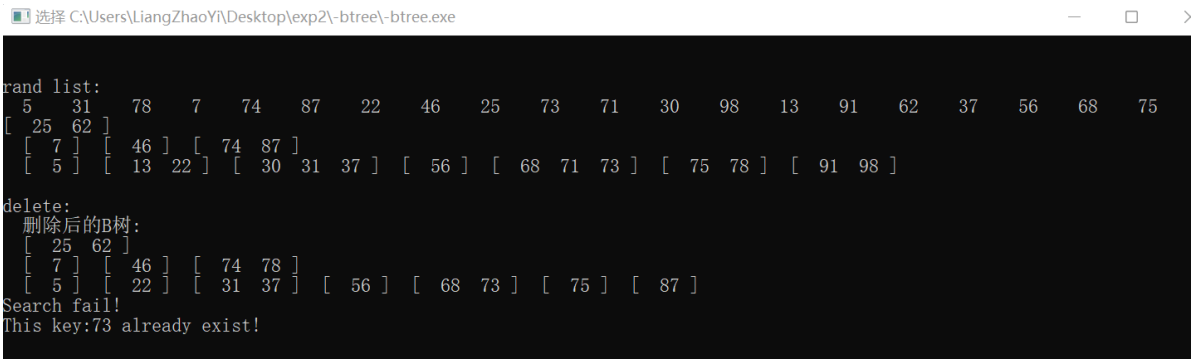
首先，生成一个大小为100，范围为0~1000的数组，插入节点并形成树。随后删除其中50~95的数，打印删除后的树。

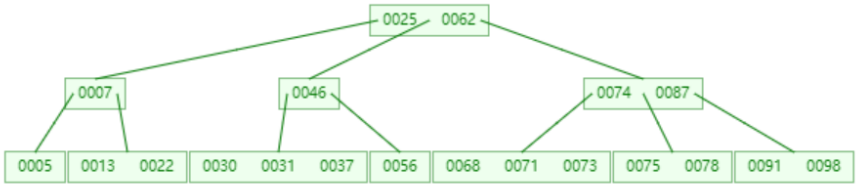


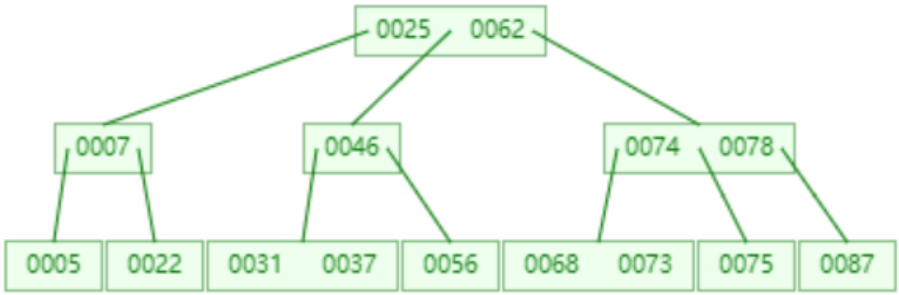
如图所示。由于b树的结构不适宜翻转后打印，因此我使用队列储存树的结构后，再打印出来。好处是可视程度更高，坏处是当树十分大时，在屏幕难以显示。在上图中，每一个开头对齐的行代表着树的一层，如图可见，树的结构在大规模数组的实现上正确。

2.使用小规模的随机数组检验树的结构。

我们使用范围为1~100，大小为20的随机数组验证数的结构是正确的。生成树并且打印，如图所示：







可以看到，检验的结果一致。并且正确实现了查找功能。

此外，在-b树的实现中使用了c++部分功能，因此main函数没有单独分开，而是放在-btree.cpp文件中实现。

主函数如下：

```
int main(){
    BTNode *t=NULL;                                          //大规模数组检验
    NODE0 s;
    int j,n=100;
    int k;
    int a[100] ;
    num_rand(a,100,1000);
    int i;
    for(i=0;i<100;i++){
        printf("%d  ",a[i] );
    }
    printf("\n\n");
    for(j=0;j<n;j++){
        s=SearchBTree(t,a[j]);
        if(s.tag==0)
            InsertBTree(t,s.i,a[j],s.pt);
    }
    PrintBTree(t);
    printf("\n");
    printf("delete:\n");
    for(i=70;i<90;i++)
        BTreeDelete(t,a[i]);
    printf("  删除后的B树: \n");
    PrintBTree(t);
    printf("\n\n");

    int b[20] ;                                                //小规模数组检验
    num_rand(b,20,100);
    printf("\nrand list:\n");
    for(i=0;i<20;i++)
        printf("  %d  ",b[i] );
```

```
    BTNode *t1=NULL;
    n=20;
    for(j=0;j<n;j++){
        s=SearchBTree(t1,b[j]);
        if(s.tag==0)
        InsertBTree(t1,s.i,b[j],s.pt);
    }
    PrintBTree(t1);
    printf("\n");
    printf("delete:\n");
    for(i=10;i<15;i++)
        BTreeDelete(t1,b[i]);
    printf("  删除后的B树: \n");
    PrintBTree(t1);

    if(SearchBTree(t1,b[10]).tag!=0)
        printf("This key: already exist!\n");
        else
        printf("Search fail!\n");
    if(SearchBTree(t1,b[9]).tag!=0)
        printf("This key:%d already exist!\n",b[9]);
        else
        printf("Search fail!\n");
    printf("\n\n");

    int c[20000] ;                                          //测量相关功能时间
    num_rand(c,20000,100000);
    n=1000;
    printf("\n\n");
    BTNode *t2=NULL;
    for(j=0;j<n;j++){
        s=SearchBTree(t2,c[j]);
        if(s.tag==0)
            InsertBTree(t2,s.i,c[j],s.pt);
    }
    n=20000;
    clock_t begin = 0, end = 0;
    begin = clock();
    for(j=1000;j<n;j++){
        s=SearchBTree(t2,c[j]);
        if(s.tag==0)
            InsertBTree(t2,s.i,c[j],s.pt);
    };
    end = clock();
    printf("\nInsert time: %5.3f\n",(float)(end-begin)/CLOCKS_PER_SEC);

    begin = clock();
    for(i=1000;i<20000;i++)
        BTreeDelete(t2,c[i]);
    end = clock();
    printf("\nDelete time: %5.3f\n",(float)(end-begin)/CLOCKS_PER_SEC);

    for(j=1000;j<n;j++){
        s=SearchBTree(t2,c[j]);
        if(s.tag==0)
            InsertBTree(t2,s.i,c[j],s.pt);
    };
```

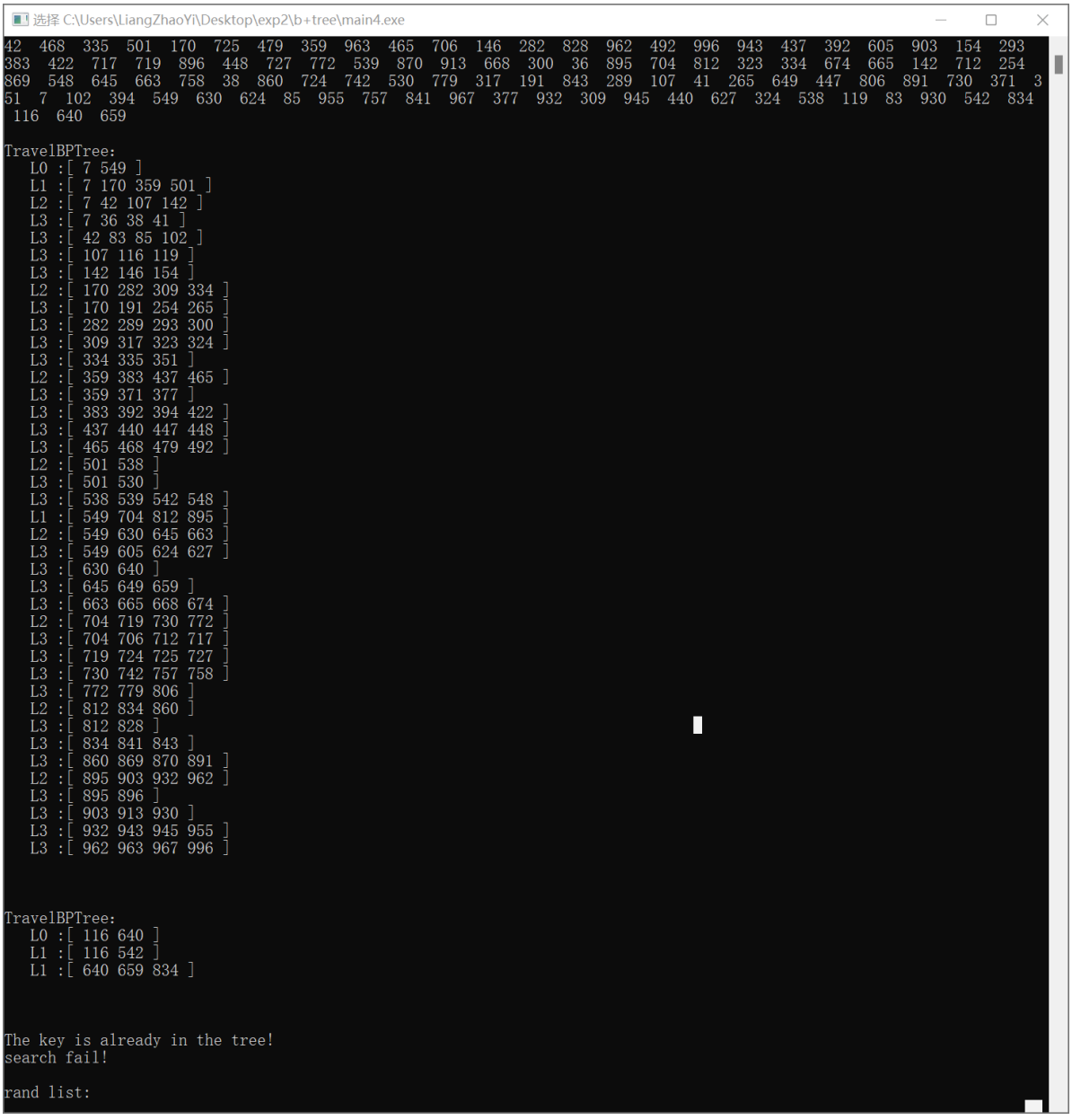
```
    begin = clock();
    for(i=1000;i<20000;i++)
        SearchBTree(t2,c[j]);
    end = clock();
    printf("\nSearch time: %5.3f\n",(float)(end-begin)/CLOCKS_PER_SEC);
    return 0;
}
```

(5) 实验内容

给定一个包含n个元素 的实数数组，请构建一颗B+树，并实现节点的插入、删除和搜索过程；

1.使用大的随机数组进行验证。

首先，生成一个大小为100，范围为0~1000的数组，插入节点并形成树。随后删除其中0~95的数，打印删除后的树。



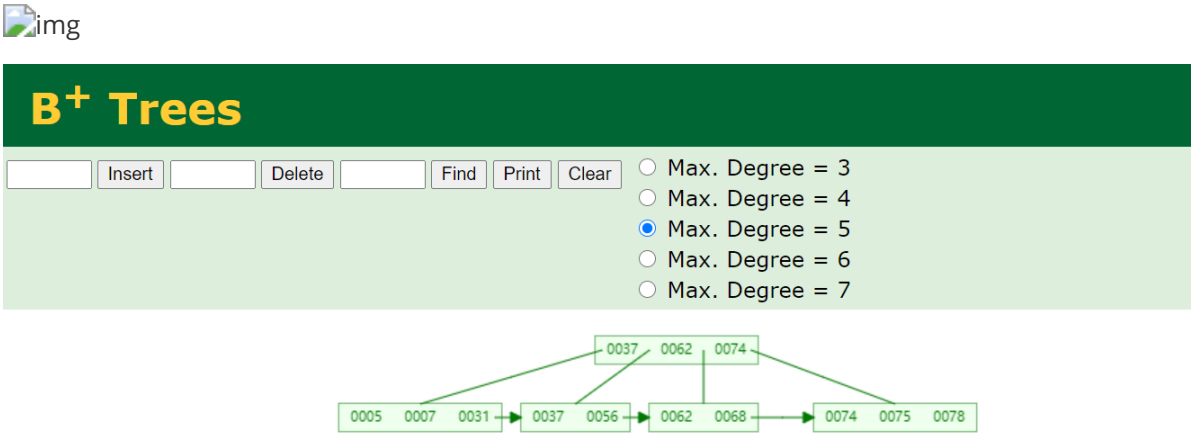
如图所示，树的结构正确，删除可以正确实现，并且查找功能也可以完成。

树的结构是通过先序遍历打印，且L*代表树的层数，这样打印也比较直观。

2.使用小规模的随机数组检验树的结构。

我们使用范围为1~100，大小为20的随机数组验证数的结构是正确的。生成树并且打印，如图所示

```
rand list:
  5    31    78    7    74    87    22    46    25    73    71    30    98    13    91    62    37    56    68    75

TravelBPTree:
          L0 :[ 5 37 ]
   L1 :[ 5 13 30 ]
   L2 :[ 5 7 ]
   L2 :[ 13 22 25 ]
   L2 :[ 30 31 ]
   L1 :[ 37 68 73 78 ]
   L2 :[ 37 46 56 62 ]
   L2 :[ 68 71 ]
   L2 :[ 73 74 75 ]
   L2 :[ 78 87 91 98 ]


--------------------------------------------------------------------------------
   L0 :[ 5 37 ]
   L1 :[ 5 13 30 ]
   L2 :[ 5 7 ]
   L2 :[ 13 22 25 ]
   L2 :[ 30 31 ]
   L1 :[ 37 68 73 78 ]
   L2 :[ 37 46 56 62 ]
   L2 :[ 68 71 ]
   L2 :[ 73 74 75 ]
   L2 :[ 78 87 91 98 ]


--------------------------------------------------------------------------------
after delete:
   L0 :[ 5 62 ]
   L1 :[ 5 37 ]
   L2 :[ 5 7 31 ]
   L2 :[ 37 56 ]
   L1 :[ 62 74 ]
   L2 :[ 62 68 ]
   L2 :[ 74 75 78 ]


--------------------------------------------------------------------------------
```



由结果可见，本人代码实现的b+树与网页上结构大相径庭，但是叶子节点得到的序列是一样的，说明二者的功能都可以实现。事实上，网页上给出的b+树与老师PPT上的也不同。b+树内点孩子节点的指向不同，自然实现的结构不同。在我的b+树中，内点指向子节点关键码的最小值，而课本上是指向关键码的最大值，二者在检索时无非是从大到小和从大到小的区别，在c程序中功能是一致的。个人推测，由于b+树常用于地址的存放，内点的孩子指向子节点关键码的最小值的模式更有利于向高地址增长的存储模式。

主函数代码：

```
int main()

{   int a[100] ;                                                    //大规模数组检验
    num_rand(a,100,1000);
    int i;
    for(i=0;i<100;i++){
```

```
        printf("%d  ",a[i] );
    }
    printf("\n\n");
    BPTree T;
    T = InitBPTree();

    for (i =0; i<100; i++) {
        T = InsertBPTree(T, a[i]);
    }

    printf("TravelBPTree: \n");
    TravelBPTree(T);
    printf("\n\n");

    for(i=0;i<95;i++)
        T = RemoveBPTree(T, a[i]);
    printf("TravelBPTree: \n");
    TravelBPTree(T);
    printf("\n\n");

    if(SearchBPTree(T, 116)!=1)
        printf("search fail!\n");
        else
        printf("The key is already in the tree!\n");
        if(SearchBPTree(T, 980)!=1)
        printf("search fail!\n");
    DeleteBPTree(T);

    int b[20] ;                                          //小规模数组检验
    num_rand(b,20,100);
    printf("\nrand list:\n");
    for(i=0;i<20;i++)
        printf("  %d  ",b[i] );
    BPTree T1;
    T1 = InitBPTree();

    for (i = 0; i < 20; i++) {
        T1 = InsertBPTree(T1, b[i]);
    }
    printf("\n\n");
    printf("TravelBPTree: \n\t");
    TravelBPTree(T1);
    printf("\n_____________________________________________________________________\n");

    TravelBPTree(T1);
    printf("\n_____________________________________________________________________\n");
    for(i=5;i<15;i++)
        T1 = RemoveBPTree(T1, b[i]) ;
    printf("after delete:\n");
    TravelBPTree(T1);
    printf("\n_____________________________________________________________________\n");

    int c[20000] ;                                      //测量各功能实现的时间
    num_rand(c,20000,100000);

    printf("\n\n");
    BPTree T2;
    T2 = InitBPTree();
```

```
    for (i =0; i<1000; i++) {
        T2 = InsertBPTree(T2, c[i]);
    }
    clock_t begin = 0, end = 0;
    begin = clock();
    for(i=1000;i<20000;i++)
        InsertBPTree(T2, c[i]);
    end = clock();
    printf("\nInsert time: %5.3f\n",(float)(end-begin)/CLOCKS_PER_SEC);

    begin = clock();
    for(i=1000;i<20000;i++)
        T2 = RemoveBPTree(T2, c[i]);
    end = clock();
    printf("\nDelete time: %5.3f\n",(float)(end-begin)/CLOCKS_PER_SEC);

    for(i=1000;i<20000;i++)
        InsertBPTree(T2, c[i]);

    begin = clock();
    for(i=1000;i<20000;i++)
        SearchBPTree(T2, c[i]);
    end = clock();
    printf("\nSearch time: %5.3f\n",(float)(end-begin)/CLOCKS_PER_SEC);
    return 0;
}
```

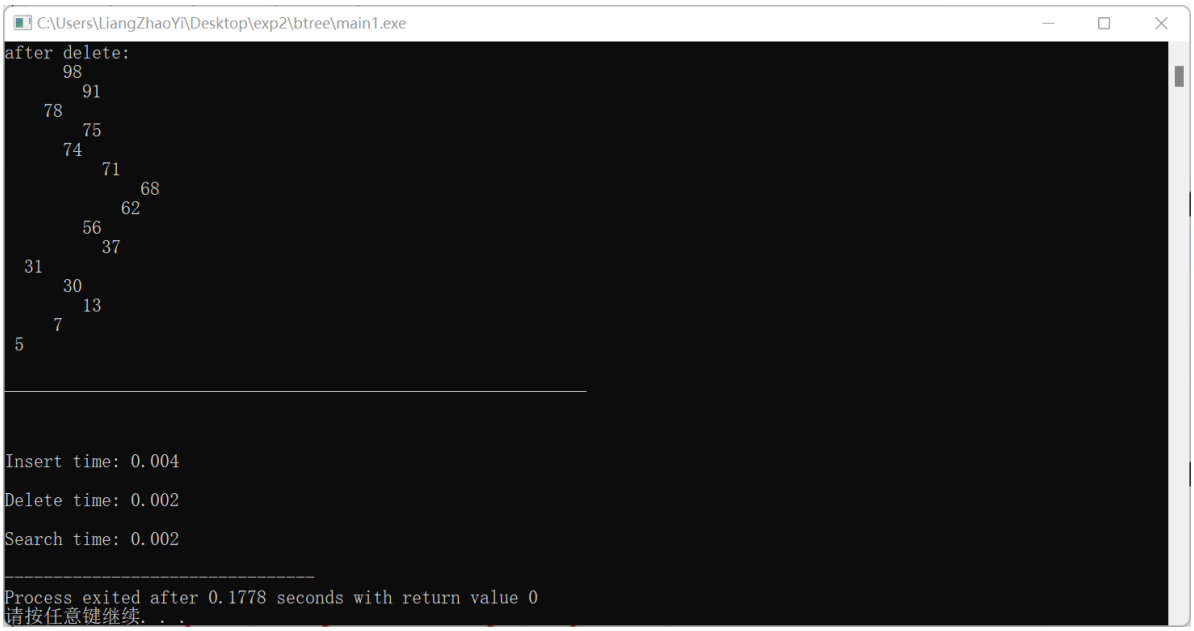
（6）实验内容

随机生成一个包含n个元素的实数数组，请比较在二叉搜索树、AVL树、红黑树、5阶-B树、5阶-B+树上进行相同节点插入、删除和搜索三个过程时间复杂度。

代码实现：

主函数的代码如上文所示。我们先构造一个大小为1000的树。由于系统时钟的精度有限，而数组大小受到rand函数生成随机数范围的限制，我们尽量扩大数的范围，从而保证统计的精度。在已有树的基础上实现1000~20000，共19000个数的插入、删除以及搜索。统计时间，结果如下：

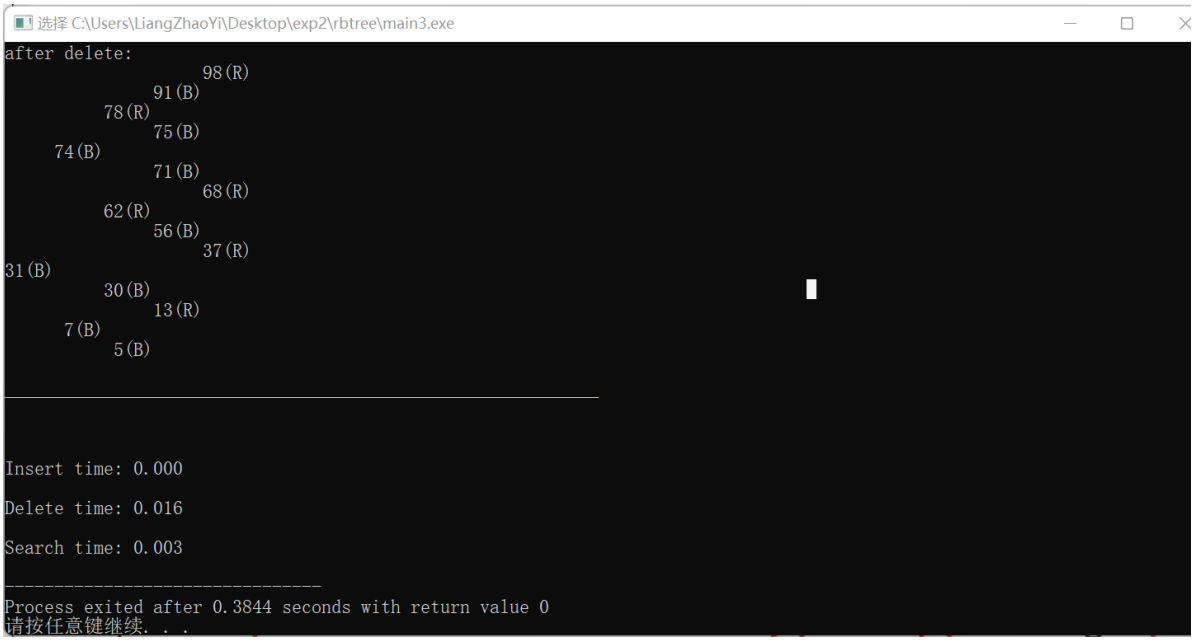
二叉查找树：



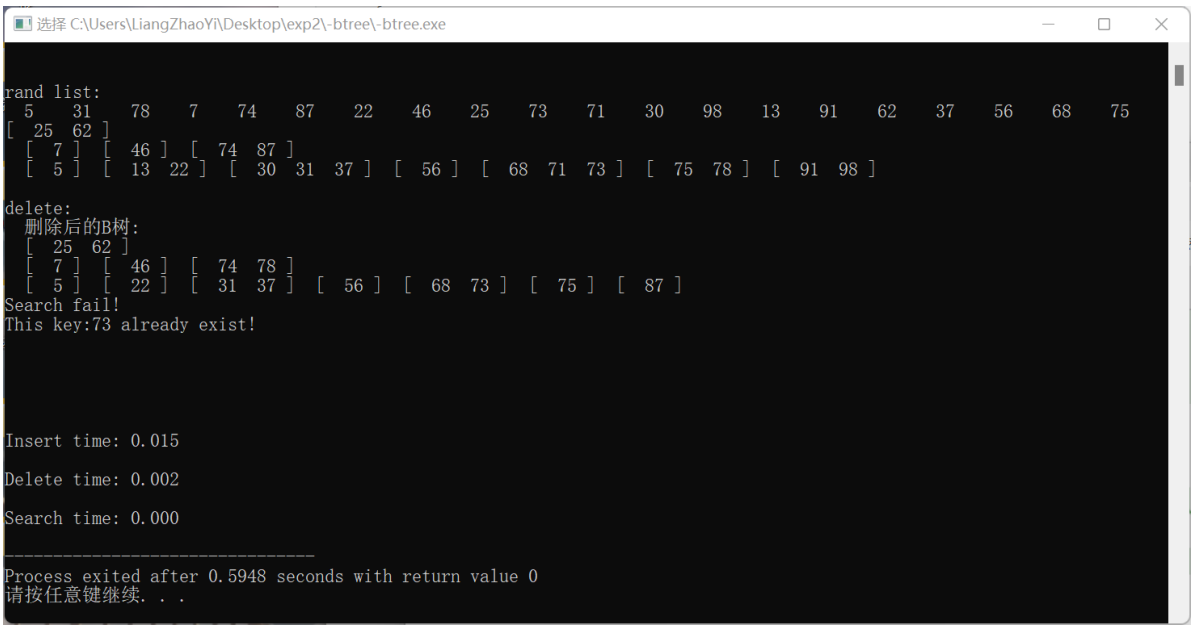
AVL树：


C:\Users\LiangZhaoYi\Desktop\exp2\avltree\main2.exe
after delete:
98
91
78
75
74
71
68
62
56
37
31
30
13
7
5
Insert time: 0.015
Delete time: 0.002
Search time: 0.000
Process exited after 0.5269 seconds with return value 0
请按任意键继续. . .

红黑树：


选择 C:\Users\LiangZhaoYi\Desktop\exp2\rbtree\main3.exe
after delete:
98(R)
91(B)
78(R)
75(B)
74(B)
71(B)
68(R)
62(R)
56(B)
37(R)
31(B)
30(B)
13(R)
7(B)
5(B)
Insert time: 0.000
Delete time: 0.016
Search time: 0.003
Process exited after 0.3844 seconds with return value 0
请按任意键继续. . .

-B树：



B+树:

tree\time	Insert time	Delete time	Search time
BINTree	0.004	0.002	0.002
AVLTree	0.015	0.002	0.000
RBTree	0.000	0.015	0.003
-BTree	0.015	0.002	0.000
B+Tree	0.000	0.016	0.078

如图可见，算法需要的时间不稳定。首先，各个功能需要的时间与树的结构挂钩，例如二叉查找树，其插入和删除不需要选择操作，时间复杂度自然较小，且与插入顺序与树的结构有关。

事实上，各种功能需要的时间也与算法的实现方式有关。例如我在-B树和B+树的搜索运算中，-B树使用了非递归的方法，且设置了标志，减少了运算的冗余，而B+树使用了递归算法，而且由于数据全部保存在叶子节点上，每次都要遍历整棵树，其时间复杂度自然更大。

-B树和B+树搜索的代码如下：

```
//-B树非递归实现查找
int SearchBTNode(BTNode *p,KeyType k){                //查找子节点是否有该关键字
    int i=0;
    for(i=0;i<p->keynum&&p->key[i+1]<=k;i++);
    return i;
}


NODE0 SearchBTree(BTree t,KeyType k){

    BTNode *p=t,*q=NULL;                                点,q指向p的双亲
    int found_tag=0;                                     //设定查找成功与否标志
    int i=0;
    NODE0 r;                                            //设定返回的查找结果

    while(p!=NULL&&found_tag==0){
        i=SearchBTNode(p,k);
        if(i>0&&p->key[i]==k)
            found_tag=1;
        else{
            q=p;
            p=p->ptr[i];
        }
    }

    if(found_tag==1){
        r.pt=p;
        r.i=i;
        r.tag=1;
    }
    else{
        r.pt=q;
        r.i=i;
        r.tag=0;
    }
    return r;                                            //返回关键字k的位置(或插入位置)
}

//B+树递归实现查找
int SearchBPTree(BPTree T,int key){
    int i;
    if (T != NULL){
        if (T->Children[0] == NULL)
            i = 0;
        else
            i = 1;

        while (i < T->KeyNum)
        if(key==T->Key[i++]){
            return 1;
            }

        i = 0;
        while (i <= T->KeyNum) {
            if(SearchBPTree(T->Children[i], key)==1)
            return 1;
            i++;
```

```
        }
    }
}
```

实验总结

这次实验，我实现了五种不同树的插入，删除以及搜索功能，加深对不同树的了解。由于这次实验的代码量巨大，我的编程能力也得到了相应的提升。在实验中，树的删除操作是一大难点，很容易陷入死循环以及莫名其妙删掉其他节点，为此我进行了大量的测试以及debug。此外，网站上的树与我生成的部分树在删除后，以及结构上都有出入，为此我理解代码代码，与网站上动画对比，发现其中的不同。总的来说，这次实验收获满满，就是量有点大，足足消化了两个星期。