《数据结构与算法实验》第 13 次实验

学院： 专业： 年级：
姓名： 学号 日期： 2022 年 6 月 16 日

第一部分 验证实验

一、 实验目的

1. 掌握顺序查找算法的基本思想，实现方法，时间性能。
2. 掌握顺序查找算法的基本思想，实现方法，时间性能。

二、 实验内容

1. 顺序查找的实现（参考实验书 p231）

- 对给定的查找集合，顺序查找与给定值 k 相等的元素。

2. 折半查找的实现（参考实验书 p232）

- 对给定的有序查找集合，折半查找与给定值 k 相等的元素。

三、代码实现（注：实验编码格式为 UTF-8）

1.SeqSearch

```
#include <iostream>
using namespace std;
#include <stdlib.h>
#include <time.h>

const int Max = 10;
int a[Max + 1] = {0};
void Creat();
int SeqSearch(int r[], int n, int k, int &count);

int main()
{
```

```
    int location = 0, count = 0, k;
    Creat();
    for (int i = 1; i <= Max; i++)
        cout<<a[i]<<" ";
    cout<<endl;
    k = 1 + rand() % Max;
    location = SeqSearch(a, Max, k, count);
    cout<<"元素"<<k<<"在序列中的序号是"<<location;
    cout<<", 共比较"<<count<<"次"<<endl;
    return 0;
}

void Creat()
{
    srand(time(NULL));
    for (int i = 1; i <= Max; i++)
        a[i] = 1 + rand() % Max;
}

int SeqSearch(int r[], int n, int k, int &count)
{
    int i = n;
    r[0] = k;
    while (++count && r[i] != k)
        i--;
    return i;
}
```

2. BinSearch

```
#include <iostream>
using namespace std;
#include <stdlib.h>
#include <time.h>

const int Max = 10;
int a[Max + 1] = {0};
void Creat();
int BinSearch(int r[], int n, int k, int &count);

int main()
{
    int location = 0, count = 0, k;
    Creat();
```

```
    for (int i = 1; i <= Max; i++)
        cout<<a[i]<<" ";
    cout<<endl;
    k = a [1 + rand() % Max];
    location = BinSearch(a, Max, k, count);
    cout<<"元素"<<k<<"在序列中的序号是"<<location;
    cout<<"，共比较"<<count<<"次"<<endl;
    return 0;
}

void Creat()
{
    srand(time(NULL));
    a[0] = 0;
    for (int i = 1; i <= Max; i++)
        a[i] = a[i-1] + rand() % Max;
}

int BinSearch(int r[], int n, int k, int &count)
{
    int low = 1, high = n;
    int mid;
    while (low <= high)
    {
        mid = (low + high)/2;
        count++;
        if (k < r[mid]) high = mid - 1;
        else return mid;
    }
    return 0;
}
```

四、运行与测试

1. SeqSearch

```
6 8 2 9 4 6 2 6 6 3
元素2在序列中的序号是7，共比较4次
PS F:\DataStructure>
```

2. BinSearch

```
7 12 14 17 20 20 26 29 32 32
元素12在序列中的序号是2，共比较2次
PS F:\DataStructure>
```

第二部分 设计实验：有向无环图

一、实验目的

通过 2 个实验，实现拓扑排序算法 TopSort，解决 AOE 网与关键路径问题。

二、实验内容

1. 拓扑排序：参考课本 p175

拓扑排序设 G=(V,E)是一个有向图，V 中的顶点序列 $v_0, v_1, \cdots, v_{n-1}$ 称为一个拓扑序列，当且仅当满足下列条件：若从顶点 v_i 到 v_j 有一条路径，则在顶点序列中顶点 v_i 必在 v_j 之前。对一个有向图构造拓扑序列的过程称为拓扑排序。

(1)从 AOV 网中选择一个没有前驱的顶点并且输出它；

(2)从 AOV 网中删去该顶点，并且删去所有以该顶点为尾的弧；

(3)重复上述两步，直到全部顶点都被输出，或 AOV 网中不存在没有前驱的顶点。

显然，拓扑排序的结果有两种：AOV 网中全部顶点都被输出，这说明 AOV 网中不存在回路；AOV 网中顶点未被全部输出，剩余的顶点均不存在没有前驱的顶点，这说明 AOV 网中存在回路。

编写伪代码如下：

1. 栈 s 初始化；累加器 count 初始化；

2. 扫描顶点表,将没有前驱（入度为 0）的顶点压栈；

3. 当栈 s 非空时循环

  3.1 j=栈顶元素出栈；输出顶点 j；count++；

  3.2 对顶点 j 的每一个邻接点 k 执行下述操作：

3.2.1 将顶点 k 的入度减 1；

3.2.2 如果顶点 k 的入度为 0，将顶点 k 入栈；

4. if (count<vertexNum)输出有回路信息；

2. AOE 网与关键路径：参考课本 p179

AOE 网具有以下两个性质：

- 只有在进入某顶点的各活动都已经结束，该顶点所代表的事件才能发生；
- 只有在某顶点所代表的事件发生后，从该顶点出发的各活动才能开始。

如果用 AOE 网来表示一项工程，那么，仅仅考虑各个活动之间的优先关系还不够，更多的是关心整个工程完成的最短时间是多少；哪些活动的延期将会影响整个工程的进度，而加速这些活动是否会提高整个工程的效率。

由于 AOE 网络中的某些活动能够同时进行，故完成整个工程必须花费的时间应该为始点到终点的最大路径长度。具有最大路径长度的路径成为关键路径，关键路径上的活动成为关键活动。关键路径长度是整个工程所需的最短工期。也就是说，要缩短整个工期，必须加快关键活动的进度。

利用 AOE 网进行工程管理时需要解决的主要问题是：

- 计算完成整个工程的最短工期；
- 确定关键路径，以找出哪些活动是影响工程进度的关键。

三、设计编码（注：实验编码格式为 UTF-8/GB2312）

1. 拓扑排序

(1)ALGraph.h

```
#ifndef ALGraph_H
#define ALGraph_H
const int MaxSize = 10;

struct ArcNode
{
    int adjvex;
    ArcNode * next;
};
```

```
template <class T>
struct VertexNode
{
    int in;
    T vertex;
    ArcNode * firstedge;
};

template <class T>
class ALGraph
{
    private:
        VertexNode<T> adjlist[MaxSize];
        int vertexNum, arcNum;
    public:
        ALGraph(T a[], int n, int e);
        ~ALGraph();
        void DFSTraverse(int v);
        void BFSTraverse(int v);
        template <class DataType>
        friend void TopSort(ALGraph<DataType> &G);
};

#endif
```

(2) ALGraph. cpp

```
#include <iostream>
#include "ALGraph.h"
using namespace std;
extern int visited[];

template <class T>
ALGraph<T>::ALGraph(T a[], int n, int e)
{
    ArcNode * s;
    int i, j, k;
    vertexNum = n;
    arcNum = e;
    for ( i = 0; i < vertexNum; i++)
    {
        adjlist[i].vertex = a[i];
        adjlist[i].firstedge = NULL;
```

```
        adjlist[i].in = 0;
    }
    for ( k = 0; k < arcNum; k++)
    {
        cout<<"请输入边的两个顶点的序号（保持序关系）：";
        cin>>i>>j;
        if (i == j)  break;
        s = new ArcNode;
        s -> adjvex = j;
        s -> next = adjlist[i].firstedge;
        adjlist[i].firstedge = s;
        adjlist[j].in++;
    }
}

template <class T>
ALGraph<T>::~ALGraph()
{
    ArcNode * p;
    for (int i = 0; i < vertexNum; i++)
    {
        p = adjlist[i].firstedge;
        while (p != NULL)
        {
            adjlist[i].firstedge = p -> next;
            delete p;
            p = adjlist[i].firstedge;
        }
    }
}

template <class T>
void ALGraph<T>::DFSTraverse(int v)
{
    ArcNode * p = NULL;
    int j;
    cout<<adjlist[v].vertex;
    visited[v] = 1;
    p = adjlist[v].firstedge;
    while (p != NULL)
    {
        j = p -> adjvex;
        if (visited[j] == 0)  DFSTraverse(j);
        p = p -> next;
```

```
    }
}

template <class T>
void ALGraph<T>::BFSTraverse(int v)
{
    int Q[MaxSize];
    int front = -1, rear = -1;
    ArcNode * p;
    cout<<adjlist[v].vertex;
    visited[v] = 1;
    Q[++rear] = v;
    while (front != rear)
    {
        v = Q[++front];
        p = adjlist[v].firstedge;
        while (p != NULL)
        {
            int j = p -> adjvex;
            if (visited[j] == 0)
            {
                cout<<adjlist[j].vertex;
                visited[j] = 1;
                Q[++rear] = j;
            }
            p = p -> next;
        }
    }
}

template <class DataType>
void TopSort(ALGraph<DataType> &G)
{
    int count = 0, top = -1;
    for (int i = 0; i < G.vertexNum; i++)
    {
        if (G.adjlist[i].in == 0)
        {
            G.adjlist[i].in = top;
            top = i;
        }
    }
    while (top != -1)
    {
```

```
        int j = top;
        top = G.adjlist[top].in;
        cout<<G.adjlist[j].vertex<<" ";
        count++;
        ArcNode * p = G.adjlist[j].firstedge;
        while (p != NULL)
        {
            int k = p -> adjvex;
            G.adjlist[k].in--;
            if (G.adjlist[k].in == 0)
            {
                G.adjlist[k].in = top;
                top = k;
            }
            p = p->next;
        }
    }
    if (count < G.vertexNum)  cout<<"有回路"<<endl;
}
```

(3)ALGraph_main.cpp

```
#include <iostream>
#include"ALGraph.cpp"
using namespace std;
int visited[MaxSize] = {0};

int main()
{
    string ch[] = {"A", "B", "C", "D", "E", "F"};
    int i;
    ALGraph<string> ALG(ch, 6, 9);
    for (int i = 0; i < MaxSize; i++)  visited[i] = 0;
    TopSort(ALG);
    return 0;
}
```

2. AOE 网与关键路径

(1)MGraph.h

```
#ifndef MGraph_H
#define MGraph_H

const int MaxSize = 10;
const int INF = 1000;
extern int visited[MaxSize];

template <class DataType>
class MGraph
{
    public:
        MGraph(DataType a[], int n, int e);
        ~MGraph();
        void DFSTraverse(int v);
        void BFSTraverse(int v);
        template <class T>
        friend int GetVL(MGraph<T> &G, int k, int vertexTop[], int vl[]);
        template <class T>
        friend int GetVE(MGraph<T> &G, int k, int vertexTop[], int ve[]);
        template <class T>
        friend void AOE(MGraph<T> &G);
    private:
        DataType vertex[MaxSize];
        int arc[MaxSize][MaxSize];
        int vertexNum, arcNum;
};

#endif
```

(2)MGraph.cpp

```
#include <iostream>
#include "MGraph.h"
using namespace std;

extern int visited[];

template <class DataType>
MGraph<DataType>::MGraph(DataType a[], int n, int e)
{
    int i, j, m;
```

```
    vertexNum = n;
    arcNum = e;
    for ( i = 0; i < vertexNum; i++)
        vertex[i] = a[i];
    for ( i = 0; i < vertexNum; i++)
        for (j = 0; j < vertexNum; j++)
            arc[i][j] = 0;
    for (int k = 0; k < arcNum; k++)
    {
        cout<<"请输入边的两个顶点的序号和其权值：";
        cin>>i;
        cin>>j;
        cin>>m;
        arc[i][j] = m;
    }
}

template <class DataType>
MGraph<DataType>::~MGraph()
{
    //empty
}

template <class DataType>
void MGraph<DataType>::DFSTraverse(int v)
{
    cout<<vertex[v];
    visited[v] = 1;
    for (int j = 0; j < vertexNum; j++)
    {
        if (arc[v][j] == 1 && visited[j] == 0)
            DFSTraverse(j);
    }
}

template <class DataType>
void MGraph<DataType>::BFSTraverse(int v)
{
    int Q[MaxSize];
    int front = -1, rear = -1;
    cout<<vertex[v];
    visited[v] = 1;
    Q[++rear] = v;
    while (front != rear)
```

```
    {
        v = Q[++front];
        for (int j = 0; j < vertexNum; j++)
            if (arc[v][j] == 1 && visited[j] == 0)
            {
                cout<<vertex[j];
                visited[j] = 1;
                Q[++rear] = j;
            }
    }
}

//计算最早发生时间
template <class T>
int GetVE(MGraph<T> &G, int k, int vertexTop[], int ve[])
{
    int max = -1, i;
    for ( i = 0; i < G.vertexNum; i++)
    {
        if (G.arc[i][k])
        {
            if (ve[vertexTop[i]] + G.arc[i][k] > max)
                max = ve[vertexTop[i]] + G.arc[i][k];
        }
    }
    if (max == -1)  return 0;
    else  return max;
}

//计算最迟发生时间
template <class T>
int GetVL(MGraph<T> &G, int k, int vertexTop[], int vl[])
{
    int min = INF, i;
    for ( i = 0; i < G.vertexNum; i++)
    {
        if (G.arc[k][i])
        {
            if (vl[vertexTop[i]] - G.arc[k][i] < min)
                min = vl[vertexTop[i]] - G.arc[k][i];
        }
    }
    return min;
}
```

```
template <class T>
void AOE(MGraph<T> &G)
{
    int vertexTop[G.vertexNum];
    int S1[G.vertexNum], S2[G.vertexNum];
    int ve[G.vertexNum], vl[G.vertexNum];
    int ee[G.vertexNum][G.vertexNum] = {-1};
    int el[G.vertexNum][G.vertexNum] = {-1};
    int i, j, count = 0, top1 = -1, top2 = -1;
    int in[G.vertexNum] = {0};
    for ( j = 0; j < G.vertexNum; j++)
        for ( i = 0; i < G.vertexNum; i++)
            if (G.arc[i][j])  in[j]++;
    for ( i = 0; i < G.vertexNum; i++)
        if (!in[i])  S1[++top1] = i;
    while (top1 != -1)
    {
        int tmp = S1[top1--];
        vertexTop[tmp] = count;
        ve[count] = GetVE(G, tmp, vertexTop, ve);
        count++;
        S2[++top2] = tmp;
        for ( j = 0; j < G.vertexNum; j++)
        {
            if (G.arc[tmp][j])
            {
                in[j]--;
                if (!in[j])  S1[++top1] = j;
            }
        }
    }
    if (count < G.vertexNum)
    {
        cout<<"有回路"<<endl;
        return;
    }
    count--;
    int temp = S2[top2--];
    vl[count] = ve[count];
    while (top2 != -1)
    {
        count--;
        int tmp = S2[top2--];
```

```
        vl[count] = GetVL(G, tmp, vertexTop, vl);
    }
    // 计算 ee
    for (i = 0; i < G.vertexNum; i++)
        for (j = 0; j < G.vertexNum; j++)
        {
            if (G.arc[i][j])
                ee[i][j] = ve[vertexTop[i]];
        }
    // 计算 el
    cout << "关键活动有:" << endl;
    for (i = 0; i < G.vertexNum; i++)
        for (j = 0; j < G.vertexNum; j++)
            if (G.arc[i][j])
            {
                el[i][j] = vl[vertexTop[j]] - G.arc[i][j];
                if (el[i][j] == ee[i][j])
                {
                    cout << G.vertex[i] << " -> " << G.vertex[j] << endl;
                }
            }
}
```

(3)MGraph_main.cpp

```
#include <iostream>
#include <string.h>
using namespace std;
#include "MGraph.cpp"
int visited[MaxSize] = {0};

int main()
{
    string ch[] = {"v0", "v1", "v2", "v3", "v4", "v5", "v6", "v7", "v8"};
    MGraph<string> MG(ch, 9, 11);
    AOE(MG);
    return 0;
}
```

四、运行结果

1. 拓扑排序

```
请输入边的两个顶点的序号（保持序关系）：1 0
请输入边的两个顶点的序号（保持序关系）：2 0
请输入边的两个顶点的序号（保持序关系）：2 3
请输入边的两个顶点的序号（保持序关系）：1 3
请输入边的两个顶点的序号（保持序关系）：3 0
请输入边的两个顶点的序号（保持序关系）：4 2
请输入边的两个顶点的序号（保持序关系）：4 5
请输入边的两个顶点的序号（保持序关系）：4 3
请输入边的两个顶点的序号（保持序关系）：3 5
E C B D A F
```

2. AOE 网和关键路径

```
请输入边的两个顶点的序号和其权值：0 1 6
请输入边的两个顶点的序号和其权值：0 2 4
请输入边的两个顶点的序号和其权值：0 3 5
请输入边的两个顶点的序号和其权值：1 4 1
请输入边的两个顶点的序号和其权值：2 4 1
请输入边的两个顶点的序号和其权值：3 5 2
请输入边的两个顶点的序号和其权值：4 6 9
请输入边的两个顶点的序号和其权值：4 7 7
请输入边的两个顶点的序号和其权值：5 7 4
请输入边的两个顶点的序号和其权值：6 8 2
请输入边的两个顶点的序号和其权值：7 8 4
关键活动有：
v0 -> v1
v1 -> v4
v4 -> v6
v4 -> v7
v6 -> v8
v7 -> v8
```

第三部分 设计实验

一、实验目的

通过 5 个实验，熟练掌握查找技术的相关代码操作。

二、实验内容

1. **顺序查找（哨兵在高端）**：参考课本 p217

- 设计顺序查找算法，将哨兵设在下标的高端。

• 将哨兵设在在下标高端，表示从数组的低端开始查找，在查找不成功的情况下，算法自动在哨兵处终止。

2. 二叉排序树所在层数：参考课本 p217

• 编写算法求给定结点在二叉排序树中所在的层数。

• 根据题目要求采用递归方法，从根结点开始查找结点 p，若待查结点深度为 1，否则到左子树（或右子树）上去查找，查找深度加 1。

3. 最近公共祖先：参考课本 p217

• 编写算法，在二叉排序树上找出任意两个不同结点的最近公共祖先。

• 设两个结点分别为 A 和 B, 根据题目要求分下面情况讨论:

(1)若 A 为根结点，则 A 为公共祖先；

(2)若 A->data < root->data 且 root->data < B->data，root 为公共祖先；

(3)若 A->data < root->data 且 B->data < root->data，则到左子树查找；

(4)若 A->data > root->data 且 B->data > root->data，则到右子树查找。

4. 判定二叉排序树：参考课本 p217

设计算法判定一棵二叉树是否为二叉排序树。

对二叉排序树来讲，其中序遍历序列为一个递增序列。因此，对给定二叉树进行中序遍历，如果始终能够保证前一个值比后一个值小，则说明该二叉树是二叉排序树。

5. 二叉排序树的查找性能：参考实验书 p235

• 对给定的同一个查找集合，按升序和随机顺序建立两棵二叉排序树；

• 比较同一个待查值在不同二叉排序树，上进行查找的比较次数；

• 对随机顺序建立的二叉排序树，输出查找的最坏情况和平均情况。

三、设计编码（注：实验编码格式为 UTF-8/GB2312）

1. 顺序查找（哨兵在高端）

```
#include <iostream>
using namespace std;
#include <stdlib.h>
#include <time.h>

const int Max = 10;

int a[Max + 1] = {0};
void Creat();
int SeqSearch(int r[], int n, int k, int &count);

int main()
{
    int location = 0, count = 0, k;
    Creat();
    for (int i = 1; i <= Max; i++)
        cout<<a[i]<<" ";
    cout<<endl;
    k = 1 + rand() % Max;
    location = SeqSearch(a, Max, k, count);
    cout<<"元素"<<k<<"在序列中的序号是"<<location;
    cout<<"，共比较"<<count<<"次"<<endl;
    return 0;
}

void Creat()
{
    srand(time(NULL));
    for (int i = 1; i <= Max; i++)
        a[i] = 1 + rand() % Max;
}

int SeqSearch(int r[], int n, int k, int &count)
{
    int i = n;
    r[0] = k;
    while (++count && r[i] != k)
        i--;
    return i;
}
```

2. 二叉排序树所在层数

(1)Bitree.h

```
#ifndef Bitree_H
#define Bitree_H
const int Max = 100;
const int Len = 10;

struct BiNode
{
    int data;
    BiNode *lchild, *rchild;
};

class Bitree
{
public:
    Bitree(int a[], int n);              // 建立一棵二叉排序树
    ~Bitree();                           // 析构函数
    void Insert(int k);                      // 插入数值
    void InsertBST(BiNode *&root, BiNode *s); // 插入节点
    void DeleteBST(BiNode *p, BiNode *f);     // 删除 f 的左孩子 p
    BiNode *Search(int k);                   // 搜索数值
    BiNode *SearchBST(BiNode *root, int k);
    int Level(int k);                        // 查找数值层数
    int LevelBST(BiNode *root, BiNode *p);    // 查找节点层数
    BiNode *getRoot();
    void PreOrder();
    void InOrder();
    void PostOrder();
private:
    BiNode *root;
    BiNode *Creat(BiNode *bt);
    void Release(BiNode *bt); // 封装的释放函数
    void PreOrder(BiNode *bt);
    void InOrder(BiNode *bt);
    void PostOrder(BiNode *bt);
};

#endif
```

(2)Bitree.cpp

```cpp
#include <iostream>
#include <cstdio>
using namespace std;
#include "Bitree.h"

Bitree::Bitree(int a[], int n)
{
    root = NULL;
    for (int i = 0; i < n; i++)
    {
        BiNode *s = new BiNode;
        s -> data = a[i];
        InsertBST(root, s);
    }
}

Bitree::~Bitree()
{
    Release(root);
}

void Bitree::Insert(int k)
{
    BiNode *s = new BiNode;
    s -> data = k;
    InsertBST(root, s);
}

void Bitree::InsertBST(BiNode *&root, BiNode *s)
{
    if (root == NULL)
    {
        root = s;
        root -> lchild = NULL;
        root -> rchild = NULL;
    }
    else if (s -> data < root -> data)  InsertBST(root -> lchild, s);
    else  InsertBST(root -> rchild, s);
}

void Bitree::DeleteBST(BiNode *p, BiNode *f)
{
    f -> lchild = NULL;
    Release(p);
```

```
}

BiNode *Bitree::Search(int k)
{
    return SearchBST(root, k);
}

BiNode *Bitree::SearchBST(BiNode *root, int k)
{
    if (root == NULL)  return NULL;
    else if (root -> data == k)  return root;
    else if (root -> data > k)  return SearchBST(root -> lchild, k);
    else  return SearchBST(root -> rchild, k);
}

int Bitree::Level(int k)
{
    return LevelBST(root, Search(k));
}

int Bitree::LevelBST(BiNode *root, BiNode *p)
{
    if (p == NULL)  return 0;
    if (p == root)  return 1;
    else if (root -> data > p -> data)  return LevelBST(root -> lchild, p) + 1;
    else  return LevelBST(root -> rchild, p) + 1;
}

void Bitree::Release(BiNode *bt)
{
    if (bt != NULL)
    {
        Release(bt -> lchild);
        Release(bt -> rchild);
        delete bt;
    }
}

BiNode *Bitree::getRoot()
{
    return root;
}

void Bitree::PreOrder()
```

```
{
    PreOrder(root);
}

void Bitree::InOrder()
{
    InOrder(root);
}

void Bitree::PostOrder()
{
    PostOrder(root);
}

void Bitree::PreOrder(BiNode *bt)
{
    if (bt == NULL)  return;
    else
    {
        cout<<bt -> data<<" ";
        PreOrder(bt -> lchild);
        PreOrder(bt -> rchild);
    }
}

void Bitree::InOrder(BiNode *bt)
{
    if (bt == NULL)  return;
    else
    {
        InOrder(bt -> lchild);
        cout<<bt -> data<<" ";
        InOrder(bt -> rchild);
    }
}

void Bitree::PostOrder(BiNode *bt)
{
    if (bt == NULL)  return;
    else
    {
        PostOrder(bt -> lchild);
        PostOrder(bt -> rchild);
        cout<<bt -> data<<" ";
```

```
    }
}
```

(3)Bitree_main.cpp

```
#include <iostream>
#include <stdlib.h>
#include <time.h>
using namespace std;
#include "Bitree.cpp"

int a[Len] = {0};

void Creat() // 生成随机数组
{
    srand(time(NULL));
    for (int i = 0; i < Len; i++)
    {
        a[i] = 1 + rand() % Max;
        for (int j = 0; j < i; j++)
            if (a[i] == a[j])  a[i]++;
    }
}

int main()
{
    Creat();
    for (int i = 0; i < Len; i++)  cout<<a[i]<<" ";
    int k = 0;
    Bitree T(a, Len);
    cout<<endl<<"中序输出树："<<endl;T.InOrder();
    cout<<endl;
    cout<<"输入查找对象："<<endl;
    cin >> k;
    BiNode *p = T.Search(k);
    if (p == NULL)  cout<<"查找失败！"<<endl;
    else  cout<<"查找对象在第"<<T.Level(k)<<"层"<<endl;
    return 0;
}
```

3. 最近公共祖先

(1)Bitree.h

```
#ifndef Bitree_H
```

```
#define Bitree_H
const int Max = 100;
const int Len = 10;

struct BiNode
{
    int data;
    BiNode *lchild, *rchild;
};

class Bitree
{
public:
    Bitree(int a[], int n);                  // 建立一棵二叉排序树
    ~Bitree();                               // 析构函数
    void Insert(int k);                      // 插入数值
    void InsertBST(BiNode *&root, BiNode *s); // 插入节点
    void DeleteBST(BiNode *p, BiNode *f);     // 删除 f 的左孩子 p
    BiNode *Search(int k);                   // 搜索数值
    BiNode *SearchBST(BiNode *root, int k);
    BiNode *getRoot();
    void PreOrder();
    void InOrder();
    void PostOrder();
    BiNode *Ancestor(int k1, int k2);
    BiNode *AncestorBST(BiNode *A, BiNode *B, BiNode *root);
private:
    BiNode *root;
    BiNode *Creat(BiNode *bt);
    void Release(BiNode *bt); // 封装的释放函数
    void PreOrder(BiNode *bt);
    void InOrder(BiNode *bt);
    void PostOrder(BiNode *bt);
};

#endif
```

(2)Bitree.cpp

```
#include <iostream>
#include <cstdio>
using namespace std;
#include "Bitree.h"
```

```
Bitree::Bitree(int a[], int n)
{
    root = NULL;
    for (int i = 0; i < n; i++)
    {
        BiNode *s = new BiNode;
        s -> data = a[i];
        InsertBST(root, s);
    }
}

Bitree::~Bitree()
{
    Release(root);
}

void Bitree::Insert(int k)
{
    BiNode *s = new BiNode;
    s -> data = k;
    InsertBST(root, s);
}

void Bitree::InsertBST(BiNode *&root, BiNode *s)
{
    if (root == NULL)
    {
        root = s;
        root -> lchild = NULL;
        root -> rchild = NULL;
    }
    else if (s -> data < root -> data)  InsertBST(root -> lchild, s);
    else  InsertBST(root -> rchild, s);
}

void Bitree::DeleteBST(BiNode *p, BiNode *f)
{
    f -> lchild = NULL;
    Release(p);
}

BiNode *Bitree::Search(int k)
{
    return SearchBST(root, k);
```

```
}

BiNode *Bitree::SearchBST(BiNode *root, int k)
{
    if (root == NULL)  return NULL;
    else if (root -> data == k)  return root;
    else if (root -> data > k)  return SearchBST(root -> lchild, k);
    else  return SearchBST(root -> rchild, k);
}

void Bitree::Release(BiNode *bt)
{
    if (bt != NULL)
    {
        Release(bt -> lchild);
        Release(bt -> rchild);
        delete bt;
    }
}

BiNode *Bitree::getRoot()
{
    return root;
}

void Bitree::PreOrder()
{
    PreOrder(root);
}

void Bitree::InOrder()
{
    InOrder(root);
}

void Bitree::PostOrder()
{
    PostOrder(root);
}

void Bitree::PreOrder(BiNode *bt)
{
    if (bt == NULL)  return;
    else
```

```
    {
        cout<<bt -> data<<" ";
        PreOrder(bt -> lchild);
        PreOrder(bt -> rchild);
    }
}

void Bitree::InOrder(BiNode *bt)
{
    if (bt == NULL)  return;
    else
    {
        InOrder(bt -> lchild);
        cout<<bt -> data<<" ";
        InOrder(bt -> rchild);
    }
}

void Bitree::PostOrder(BiNode *bt)
{
    if (bt == NULL)  return;
    else
    {
        PostOrder(bt -> lchild);
        PostOrder(bt -> rchild);
        cout<<bt -> data<<" ";
    }
}
BiNode *Bitree::Ancestor(int k1, int k2)
{
    if (k1 > k2)
    {
        int t = k2;
        k2 = k1;
        k1 = t;
    }
    return AncestorBST(Search(k1), Search(k2), root);
}

BiNode *Bitree::AncestorBST(BiNode *A, BiNode *B, BiNode *root)
{
    if (root == NULL) return NULL;
    else if ((A -> data < root -> data) && (root -> data < B -> data) || (A ->
data == root -> data))  return root;
```

```
        else if ((A -> data > root -> data) && (root -> data < B -> data))  return
AncestorBST(A, B, root -> rchild);
            else return AncestorBST(A, B, root -> lchild);
}
```

(3)Bitree_main.cpp

```
#include <iostream>
#include <stdlib.h>
#include <time.h>
using namespace std;
#include "Bitree.cpp"

int a[Len] = {0};

void Creat() // 生成随机数组
{
    srand(time(NULL));
    for (int i = 0; i < Len; i++)
    {
        a[i] = 1 + rand() % Max;
        for (int j = 0; j < i; j++)
            if (a[i] == a[j])  a[i]++;
    }
}

int main()
{
    Creat();
    for (int i = 0; i < Len; i++)  cout<<a[i]<<" ";
    int k = 0, k1, k2;
    Bitree T(a, Len);
    cout<<endl<<"中序输出树："<<endl;T.InOrder();
    cout<<endl;
    cout<<"输入 2 个查找对象："<<endl;
    cin>>k1>>k2;
    cout<<"最近公共祖先为"<<(T.Ancestor(k1, k2) -> data)<<endl;
    return 0;
}
```

4. 判定二叉排序树

(1)Bitree.h

```
#ifndef Bitree_H
```

```
#define Bitree_H
const int Max = 100;
const int Len = 10;

struct BiNode
{
    int data;
    BiNode *lchild, *rchild;
};

class Bitree
{
public:
    Bitree(int a[], int n);                    // 建立一棵二叉排序树
    ~Bitree();                                 // 析构函数
    void Insert(int k);                        // 插入数值
    void InsertBST(BiNode *&root, BiNode *s); // 插入节点
    void DeleteBST(BiNode *p, BiNode *f);     // 删除 f 的左孩子 p
    BiNode *Search(int k);                     // 搜索数值
    BiNode *SearchBST(BiNode *root, int k);
    BiNode *getRoot();
    void PreOrder();
    void InOrder();
    void PostOrder();
    BiNode *Ancestor(int k1, int k2);
    BiNode *AncestorBST(BiNode *A, BiNode *B, BiNode *root);
    int Level(int k);
    int LevelBST(BiNode *root, BiNode *p);
    int SortBiTree();
    int SortBiTreeBST(BiNode *root);

private:
    BiNode *root;
    BiNode *Creat(BiNode *bt);
    void Release(BiNode *bt); // 封装的释放函数
    void PreOrder(BiNode *bt);
    void InOrder(BiNode *bt);
    void PostOrder(BiNode *bt);
};

#endif
```

(2)Bitree.cpp

```cpp
#include <iostream>
#include <cstdio>
using namespace std;
#include "Bitree.h"

Bitree::Bitree(int a[], int n)
{
    root = NULL;
    for (int i = 0; i < n; i++)
    {
        BiNode *s = new BiNode;
        s -> data = a[i];
        InsertBST(root, s);
    }
}

Bitree::~Bitree()
{
    Release(root);
}

void Bitree::Insert(int k)
{
    BiNode *s = new BiNode;
    s -> data = k;
    InsertBST(root, s);
}

void Bitree::InsertBST(BiNode *&root, BiNode *s)
{
    if (root == NULL)
    {
        root = s;
        root -> lchild = NULL;
        root -> rchild = NULL;
    }
    else if (s -> data < root -> data)  InsertBST(root -> lchild, s);
    else  InsertBST(root -> rchild, s);
}

void Bitree::DeleteBST(BiNode *p, BiNode *f)
{
    f -> lchild = NULL;
    Release(p);
```

```
}

BiNode *Bitree::Search(int k)
{
    return SearchBST(root, k);
}

BiNode *Bitree::SearchBST(BiNode *root, int k)
{
    if (root == NULL)  return NULL;
    else if (root -> data == k)  return root;
    else if (root -> data > k)  return SearchBST(root -> lchild, k);
    else  return SearchBST(root -> rchild, k);
}

void Bitree::Release(BiNode *bt)
{
    if (bt != NULL)
    {
        Release(bt -> lchild);
        Release(bt -> rchild);
        delete bt;
    }
}

BiNode *Bitree::getRoot()
{
    return root;
}

void Bitree::PreOrder()
{
    PreOrder(root);
}

void Bitree::InOrder()
{
    InOrder(root);
}

void Bitree::PostOrder()
{
    PostOrder(root);
}
```

```
void Bitree::PreOrder(BiNode *bt)
{
    if (bt == NULL)  return;
    else
    {
        cout<<bt -> data<<" ";
        PreOrder(bt -> lchild);
        PreOrder(bt -> rchild);
    }
}

void Bitree::InOrder(BiNode *bt)
{
    if (bt == NULL)  return;
    else
    {
        InOrder(bt -> lchild);
        cout<<bt -> data<<" ";
        InOrder(bt -> rchild);
    }
}

void Bitree::PostOrder(BiNode *bt)
{
    if (bt == NULL)  return;
    else
    {
        PostOrder(bt -> lchild);
        PostOrder(bt -> rchild);
        cout<<bt -> data<<" ";
    }
}
BiNode *Bitree::Ancestor(int k1, int k2)
{
    if (k1 > k2)
    {
        int t = k2;
        k2 = k1;
        k1 = t;
    }
    return AncestorBST(Search(k1), Search(k2), root);
}
```

```
BiNode *Bitree::AncestorBST(BiNode *A, BiNode *B, BiNode *root)
{
    if (root == NULL) return NULL;
    else if ((A -> data < root -> data) && (root -> data < B -> data) || (A ->
data == root -> data))  return root;
        else if ((A -> data > root -> data) && (root -> data < B -> data))  return
AncestorBST(A, B, root -> rchild);
            else return AncestorBST(A, B, root -> lchild);
}

int Bitree::Level(int k)
{
    return LevelBST(root, Search(k));
}

int Bitree::LevelBST(BiNode *root, BiNode *p)
{
    if (p == NULL)  return 0;
    if (p == root)  return 1;
    else if (root -> data > p -> data)  return LevelBST(root -> lchild, p) + 1;
    else  return LevelBST(root -> rchild, p) + 1;
}

int Bitree::SortBiTree()
{
    return SortBiTreeBST(root);
}

int Bitree::SortBiTreeBST(BiNode *root)
{
    if (root == NULL)
        return 1;
    else
    {
        if (root->lchild != NULL)
        {
            int dl = root -> lchild -> data;
            if (root->data > root -> lchild -> data)
                return SortBiTreeBST(root -> lchild);
            else  return 0;
        }
        if (root -> rchild != NULL)
        {
            int dr = root -> rchild -> data;
```

```
            if (root -> data < root -> rchild -> data)
                return SortBiTreeBST(root -> rchild);
            else  return 0;
        }
        return 1;
    }
}
```

(3)Bitree_main.cpp

```
#include <iostream>
#include <stdlib.h>
#include <time.h>
using namespace std;
#include "Bitree.cpp"

int main()
{
    int a[10] = {18, 27, 40, 63, 42, 55, 99, 24, 91, 6};
    int n = 10, k, k1, k2;
    Bitree T(a, n);
    cout<<endl<<"中序遍历树: "<<endl;T.InOrder();
    cout<<endl;
    if (T.SortBiTree() == 1)  cout << "是二叉排序树" << endl;
    else  cout << "不是二叉排序树" << endl;
    return 0;
}
```

5. 二叉排序树的查找性能

(1)Bitree.h

```
#ifndef Bitree_H
#define Bitree_H
const int Max = 100;
const int Len = 10;

struct BiNode
{
    int data;
    BiNode *lchild, *rchild;
};

class Bitree
{
```

```
public:
    Bitree(int a[], int n);                    // 建立一棵二叉排序树
    ~Bitree();                                 // 析构函数
    void Insert(int k);                        // 插入数值
    void InsertBST(BiNode *&root, BiNode *s);  // 插入节点
    void DeleteBST(BiNode *p, BiNode *f);      // 删除 f 的左孩子 p
    BiNode *Search(int k);                     // 搜索数值
    BiNode *SearchBST(BiNode *root, int k);
    BiNode *getRoot();
    void PreOrder();
    void InOrder();
    void PostOrder();
    BiNode *Ancestor(int k1, int k2);
    BiNode *AncestorBST(BiNode *A, BiNode *B, BiNode *root);
    int Level(int k);
    int LevelBST(BiNode *root, BiNode *p);
    int SortBiTree();
    int SortBiTreeBST(BiNode *root);

private:
    BiNode *root;
    BiNode *Creat(BiNode *bt);
    void Release(BiNode *bt);
    void PreOrder(BiNode *bt);
    void InOrder(BiNode *bt);
    void PostOrder(BiNode *bt);
};

#endif
```

(2)Bitree.cpp

```
#include <iostream>
#include <cstdio>
using namespace std;
#include "Bitree.h"

Bitree::Bitree(int a[], int n)
{
    root = NULL;
    for (int i = 0; i < n; i++)
    {
        BiNode *s = new BiNode;
        s -> data = a[i];
```

```
        InsertBST(root, s);
    }
}

Bitree::~Bitree()
{
    Release(root);
}

void Bitree::Insert(int k)
{
    BiNode *s = new BiNode;
    s -> data = k;
    InsertBST(root, s);
}

void Bitree::InsertBST(BiNode *&root, BiNode *s)
{
    if (root == NULL)
    {
        root = s;
        root -> lchild = NULL;
        root -> rchild = NULL;
    }
    else if (s -> data < root -> data)  InsertBST(root -> lchild, s);
    else  InsertBST(root -> rchild, s);
}

void Bitree::DeleteBST(BiNode *p, BiNode *f)
{
    f -> lchild = NULL;
    Release(p);
}

BiNode *Bitree::Search(int k)
{
    return SearchBST(root, k);
}

BiNode *Bitree::SearchBST(BiNode *root, int k)
{
    if (root == NULL)  return NULL;
    else if (root -> data == k)  return root;
    else if (root -> data > k)  return SearchBST(root -> lchild, k);
```

```
    else  return SearchBST(root -> rchild, k);
}

void Bitree::Release(BiNode *bt)
{
    if (bt != NULL)
    {
        Release(bt -> lchild);
        Release(bt -> rchild);
        delete bt;
    }
}

BiNode *Bitree::getRoot()
{
    return root;
}

void Bitree::PreOrder()
{
    PreOrder(root);
}

void Bitree::InOrder()
{
    InOrder(root);
}

void Bitree::PostOrder()
{
    PostOrder(root);
}

void Bitree::PreOrder(BiNode *bt)
{
    if (bt == NULL)  return;
    else
    {
        cout<<bt -> data<<" ";
        PreOrder(bt -> lchild);
        PreOrder(bt -> rchild);
    }
}
```

```
void Bitree::InOrder(BiNode *bt)
{
    if (bt == NULL)  return;
    else
    {
        InOrder(bt -> lchild);
        cout<<bt -> data<<" ";
        InOrder(bt -> rchild);
    }
}

void Bitree::PostOrder(BiNode *bt)
{
    if (bt == NULL)  return;
    else
    {
        PostOrder(bt -> lchild);
        PostOrder(bt -> rchild);
        cout<<bt -> data<<" ";
    }
}
BiNode *Bitree::Ancestor(int k1, int k2)
{
    if (k1 > k2)
    {
        int t = k2;
        k2 = k1;
        k1 = t;
    }
    return AncestorBST(Search(k1), Search(k2), root);
}

BiNode *Bitree::AncestorBST(BiNode *A, BiNode *B, BiNode *root)
{
    if (root == NULL) return NULL;
    else if ((A -> data < root -> data) && (root -> data < B -> data) || (A ->
data == root -> data))  return root;
        else if ((A -> data > root -> data) && (root -> data < B -> data))  return
AncestorBST(A, B, root -> rchild);
            else return AncestorBST(A, B, root -> lchild);
}

int Bitree::Level(int k)
{
```

```
    return LevelBST(root, Search(k));
}

int Bitree::LevelBST(BiNode *root, BiNode *p)
{
    if (p == NULL)  return 0;
    if (p == root)  return 1;
    else if (root -> data > p -> data)  return LevelBST(root -> lchild, p) + 1;
    else  return LevelBST(root -> rchild, p) + 1;
}

int Bitree::SortBiTree()
{
    return SortBiTreeBST(root);
}

int Bitree::SortBiTreeBST(BiNode *root)
{
    if (root == NULL)
        return 1;
    else
    {
        if (root->lchild != NULL)
        {
            int dl = root -> lchild -> data;
            if (root->data > root -> lchild -> data)
                return SortBiTreeBST(root -> lchild);
            else  return 0;
        }
        if (root -> rchild != NULL)
        {
            int dr = root -> rchild -> data;
            if (root -> data < root -> rchild -> data)
                return SortBiTreeBST(root -> rchild);
            else  return 0;
        }
        return 1;
    }
}
```

(3)Bitree_main.cpp

```
#include <iostream>
#include <stdlib.h>
```

```cpp
#include <time.h>
using namespace std;
#include "Bitree.cpp"

const int I = 1000000;
void Bubble(int a[], int len);
void Creat(int a[], int len);

int main()
{
    int a[Len] = {0}, b[Len] = {0};
    clock_t start, end;
    start = clock();
    for (int i = 0; i < I; i++)
    {
        Creat(a, Len);
        for (int j = 0; j < Len; j++)  b[j] = a[j];
        Bubble(b, Len);
        Bitree T1(a, Len), T2(b, Len);
    }
    end = clock();
    double time = (double)(end - start) / CLOCKS_PER_SEC;

    start = clock();
    for (int i = 0; i < I; i++)
    {
        Creat(a, Len);
        for (int j = 0; j < Len; j++)  b[j] = a[j];
        Bubble(b, Len);
        Bitree T1(a, Len), T2(b, Len);
        for (int i = 0; i < Len; i++)  T1.Search(a[i]);
    }
    end = clock();
    double time1 = (double)(end - start) / CLOCKS_PER_SEC;

    start = clock();
    for (int i = 0; i < I; i++)
    {
        Creat(a, Len);
        for (int j = 0; j < Len; j++)  b[j] = a[j];
        Bubble(b, Len);
        Bitree T1(a, Len), T2(b, Len);
        for (int i = 0; i < Len; i++)  T2.Search(b[i]);
    }
```

```
    end = clock();
    double time2 = (double)(end - start) / CLOCKS_PER_SEC;

    cout<<time<<endl<<time1<<endl<<time2<<endl;
    double avg1 = (time1 - time)/I;
    double avg2 = (time2 - time)/I;

    cout<<I<<"次查找随机情况平均时间为"<<avg1<<"秒"<<endl;
    cout<<I<<"次查找最坏情况平均时间为"<<avg2<<"秒"<<endl;
    return 0;
}

void Creat(int a[], int len)
{
    srand(time(NULL));
    for (int i = 0; i < len; i++)
    {
        a[i] = 1 + rand() % 100;
        for (int j = 0; j < i; j++)
            if (a[i] == a[j])  a[i]++;
    }
}

void Bubble(int a[], int len)
{
    int temp;
    for (int j = 0; j < len; j++)
        for (int i = 1; i < len - j; i++)
        {
            if (a[i] > a[i+1])
            {
                temp = a[i];
                a[i] = a[i+1];
                a[i+1] = temp;
            }
        }
}
```

四、运行结果

1. 顺序查找（哨兵在高端）

```
10 6 10 5 1 2 10 8 4 8
元素2在序列中的序号是6，共比较5次
PS F:\DataStructure>
```

2. 二叉排序树所在层数

```
14 81 21 69 65 74 28 90 11 59
中序输出树:
11 14 21 28 59 65 69 74 81 90
输入查找对象:
11
查找对象在第2层
```

```
76 92 93 20 34 46 29 8 19 49
中序输出树:
8 19 20 29 34 46 49 76 92 93
输入查找对象:
76
查找对象在第1层
```

3. 最近公共祖先

```
38 14 7 59 41 89 8 12 26 15
中序输出树:
7 8 12 14 15 26 38 41 59 89
输入2个查找对象:
59 26
最近公共祖先为38
PS F:\DataStructure>
```

4. 判定二叉排序树

```
中序遍历树:
6 18 24 27 40 42 55 63 91 99
是二叉排序树
PS F:\DataStructure>
```

5. 二叉排序树的查找性能

```
1.435
1.528
1.477
1000000次查找随机情况平均时间为9.3e-008秒
1000000次查找最坏情况平均时间为4.2e-008秒

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

五、总结与心得

这次的实验主要针对二叉排序树实现。

在设计程序测试二叉排序树的性能时，由于已经使用了精度较高的 double 类型变量，但是由于重复的次数过少，仍然无法达到“可测”的最低标准，因此将原本设计的重复 100 次改为了重复 1000000 次，最终能够计算得到结果。但是本程序在 VSCode 环境下依旧无法运行，暂时未找到解决办法。