



太原理工大学
TAIYUAN UNIVERSITY OF TECHNOLOGY

本科实验报告

课程名称： 数据结构

实验项目： 线性结构、树形结构、图结构、查找、排序

实验地点： 行勉楼 C216

专业班级： 软件 1423 学号： 2014006193

学生姓名： 石富元

指导教师： 杨永强

2016 年 1 月 15 日

线性结构
<p>一、实验目的和要求</p> <p>本次实习的主要目的是为了使学生熟练掌握线性表的基本操作在顺序存储结构和链式存储结构上的实现，提高分析和解决问题的能力。要求仔细阅读并理解下列例题，上机通过，并观察其结果，然后独立完成后面的实习题。</p>
<p>二、实验内容和原理</p> <p>1. 设顺序表 A 中的数据元素递增有序，试写一程序，将 x 插入到顺序表的适当位置上，使该表仍然有序。</p> <p>2. 用单链表 ha 存储多项式 $A(x) = a_0 + a_1x^1 + a_2x^2 + \dots + a_nx^n$ (其中 a_i 为非零系数)，用单链表 hb 存储多项式 $B(x) = b_0 + b_1x^1 + b_2x^2 + \dots + b_mx^m$ (其中 b_i 为非零系数)，要求计算 $C(x) = A(x) + B(x)$，结果存到单链表 hc 中。试写出程序。</p> <p>3. 设有 n 个人围坐在一个圆桌周围，现从第 s 个人开始报数，数到第 m 的人出列，然后从出列的下一个入重新开始报数，数到 m 的人又出列，如此重复，直到所有的人全部出列为止。Josephus 问题是：对于任意给定的 n, m, s, 求出按出列次序得到的 n 个人员的顺序表。</p>
<p>三、主要仪器设备</p> <p>使用的计算机：笔记本或台式计算机，Visual Studio 2015</p>
<p>四、操作方法与实验步骤</p> <p>1.</p> <pre>#include<iostream> #include<cstdlib> using namespace std; const int OK = 1; const int ERROR = 0; typedef int statue; typedef char Elemtype; typedef struct LNode { //单链表节点结构 Elemtype data; struct LNode *next; }LNode, *LinkList; class Node { public: Node(Elemtype *, bool); ~Node(); statue Insert(Elemtype, int); statue Print(); int Length(); statue Revrse(); statue NewLinkList(char*); statue DellLinkList(); protected: LinkList L = NULL; }; Node::Node(Elemtype * val, bool Res = true) {</pre>

```

NewLinkList( val );
if(Res){
    Revrse();
}
}
Node::~~Node()
{
    DelLinkList();
}

statue Node::DelLinkList()
{
    LinkList del = L, p;
    for(p = L->next;p->next;p = p->next) {
        delete del;
        del = p;
    }
    delete del;
    return OK;
}

statue Node::Insert( Elemtype ins, int n )
{
    LinkList front = L; // 前驱指针
    LinkList item = L; // 本位指针
    n--;
    for(int i = 0;item->next && i < n; i++)
    {
        front = item;
        item = item->next;
    }
    front->next = new LNode;
    front->next->data = ins;
    front->next->next = item;
    return OK;
}

//Insert
statue Node::Print() { //输出单链表
    int i = 0;
    LinkList p = L;
    cout << "序号\t地址\tdata\tnext\n" << endl;
    while(p) {
        cout << i << "\t" << p << "\t" << p->data << "\t" << p->next
        << endl;
        p = p->next;
        i++;
    }
    return OK;
}

//Print
statue Node::NewLinkList( char *data ) { //创建带头单链表,P30
    if(L){

```

```

        DeLinkList();
    }
    L = new LNode;
    L->data = '?';
    L->next = NULL;
    LinkList p = L;
    for(char *i = data;*i != '\0';i++) {
        p = new LNode;
        p->data = *i;
        p->next = L->next;
        L->next = p;
    }
    return OK;
}
//NewLinkList
int Node::Length() {    //求单链表长度
    int len;
    LinkList p = L;
    for(len = 0, p = L; p->next; p = p->next, len++);
    return len;
}
//Length
statue Node::Revrs() { //单链表的就地转置,依次把每一个节点插入到新表的前面
    LinkList p = L->next;
    LinkList q = p->next;
    LinkList s = q->next;
    p->next = NULL;
    while(s->next) {
        q->next = p; p = q;
        q = s; s = s->next; //把L的元素逐个插入新表表头
    }
    q->next = p;
    s->next = q;
    L->next = s;
    return OK;
}
2.
#include <iostream>
#include <cmath>
using std::cin;
using std::cout;
using std::endl;
typedef struct PUnit{
    int    power = 0;
    double coefficient = 0;
    struct PUnit* next = NULL;
}PUnit, *ListPUnit;
typedef struct polynomial {    //单链表节点结构
    ListPUnit units = NULL;
    int length = 0;

```

```

}Polynomial;
void polynomialInit(Polynomial &P, int initLength);
double polynomialAdd(Polynomial &L1, Polynomial &L2, double x);
void polynomialPrint(Polynomial &p);
void polynomialInit(Polynomial &L, int initLength)
{
    L.units = NULL;
    L.length = initLength;
    for (int i = 0; i < initLength; ++i)
    {
        PUnit *p = new PUnit;
        cout << "Power Number:";
        cin >> p->power;
        cout << "Coefficient Number:";
        cin >> p->coefficient;
        p->next = L.units;
        L.units = p;
    }
}
void polynomialAdd(Polynomial &L1, Polynomial &L2, Polynomial &L3)
{
    int maxLength = L1.length > L2.length ? L1.length : L2.length;
    PUnit * pL1 = NULL;
    PUnit * pL2 = NULL;
    PUnit * pL3 = NULL;
    double sum = 0;
    for (int i = 0; i < maxLength; i++)
    {
        pL1 = L1.units;
        pL2 = L2.units;

        if (pL1->power > pL2->power)
        {
            L1.units = L1.units->next;
            pL1->next = L3.units;
            L3.units = pL1;
            L3.length++;
        }
        else if (pL1->power == pL2->power)
        {
            L2.units = L2.units->next;
            pL1->coefficient += pL2->coefficient;
            pL1->next = L3.units;
            L3.units = pL1;
            delete pL2;
            L3.length++;
        }
        else
        {

```

```

        L2.units = L2.units->next;
        pL2->next = L3.units;
        L3.units = pL2;
        L3.length++;
    }

}

}
}
void polynomialPrint(Polynomial &p)
{
    for (PUnit * i = p.units; i; i = i->next)
    {
        cout << i->coefficient << ":" << i->power << endl;
    }
}
int main()
{
    Polynomial ha;
    Polynomial hb;
    Polynomial hc;
    polynomialInit(ha, 1);
    polynomialPrint(ha);
    polynomialInit(hb, 2);
    polynomialPrint(hb);
    polynomialAdd(ha, hb, hc);
    polynomialPrint(hc);
}
3.
#include <iostream>
using std::cout;
using std::endl;
using std::endl;
typedef struct LNode
{
    int order;
    struct LNode * next = NULL;
} LNode, *LinkList;
void Josephus(int n, int m, int s, LinkList & L);
void initList(LinkList & L, int n);
void printList(LinkList & L, int n);
int main(int argc, char const *argv[])
{
    LinkList L = NULL;
    Josephus(10, 3, 7, L);
    //initList(L, 10);
    //printList(L, 10);
}
void initList(LinkList & L, int n)
{

```

```

L = new LNode;
L->order = n;
L->next = L;
LNode * beginning = L;
LNode * node = NULL;
for (int i = n - 1; i > 0; i--)
{
    node = new LNode;
    node->order = i;
    node->next = L;
    L = node;
    beginning->next = L;
}
}
void printList(LinkList & L, int n)
{
    LNode * node = L;
    for (int i = 0; i < n + 4; i++)
    {
        cout << node->order << endl;
        node = node->next;
    }
}
void Josephus(int n, int m, int s, LinkList & L)
{
    initList(L, n);
    LNode * node = NULL;
    LNode * prevNode = L;
    for (int i = 0; i < s - 1; i++)
    {
        prevNode = prevNode->next;
    }
    while (L)
    {
        for (int i = 0; i < m; i++)
        {
            prevNode = prevNode->next;
        }
        cout << prevNode->next->order << endl;
        node = prevNode->next;
        if (prevNode->next != prevNode->next->next)
            prevNode->next = prevNode->next->next;
        else
            break;
        delete node;
    }
}

```

五、实验数据记录和处理

Index	Address	data	next	
0	00F6E4F8	?	00F6E7D0	Coefficient Number:1
1	00F6E7D0	H	00F6E808	Power Number:1
2	00F6E808	e	00F6E290	Coefficient Number:2
3	00F6E290	1	00F6E338	Power Number:2
4	00F6E338	a	00F6E878	Coefficient Number:3
5	00F6E878	1	00F6E5A0	Power Number:3
6	00F6E5A0	o	00F6E1E8	3:3
7	00F6E1E8	W	00F6E840	2:2
8	00F6E840	o	00F6E8B0	1:1
9	00F6E8B0	r	00F6E258	Coefficient Number:1
10	00F6E258	1	00F6E300	Power Number:1
11	00F6E300	d	00000000	Coefficient Number:2
C:\Windows\system32\cmd				Power Number:2
1				Coefficient Number:3
5				Power Number:3
9				Coefficient Number:4
4				Power Number:4
10				4:4
7				3:3
6				2:2
8				1:1
3				2:1
2				4:2
请按任意键继续. . .				6:3
				4:4
				请按任意键继续. . .

六、讨论、心得

链表存储结构比顺序表更加灵活，但占用空间更多，操作更加复杂

树形结构

一、实验目的和要求

熟悉树的各种表示方法和各种遍历方式，掌握有关算法的实现，了解树在计算机科学及其它工程技术中的应用。

二、实验内容和原理

1. 编写递归算法，计算二叉树中叶子结点的数目。
2. 编写递归算法，在二叉树中求位于先序序列中第 K 个位置的结点

三、主要仪器设备

使用的计算机：笔记本或台式计算机，Visual Studio 2015

四、操作方法与实验步骤

```
#include <iostream>
const int OK = 1;
const int ERROR = 0;
typedef char TElemType;
typedef struct BiTNode
{
    TElemType data;
    struct BiTNode *lchild, *rchild;
} BiTNode, *BiTree;
```

```

#include <stack>
using namespace std;
int PrintElement(TElemType e)
{
    //printf("%c ", e);
    cout << e;
    return OK;
}
int(*Visit)(TElemType);
int CreateBiTree(BiTree & T)
{
    TElemType ch;
    cin >> ch;
    if (ch == '.') T = NULL;
    else
    {
        T = new BiTNode;
        T->data = ch;           // 生成根结点
        CreateBiTree(T->lchild); // 构造左子树
        CreateBiTree(T->rchild); // 构造右子树
    }
    return OK;
} //CreateBiTree
int countLeaves(BiTree & T)
{
    if (!T) return 0;
    if (!(T->lchild || T->rchild)) return 1;
    return countLeaves(T->lchild) + countLeaves(T->rchild);
}
int countLeavesLegacy(BiTree const &T)
{
    if (!T)
    {
        return 0;
    }
    int sum = 0;
    stack<BiTree> s;
    BiTree branch = T;
    while (branch || !s.empty())
    {
        if (branch)
        {
            s.push(branch);
            branch = branch->lchild;
        }
        else
        {
            branch = s.top();
            if (!branch->lchild && !branch->rchild)

```

```

        {
            sum++;
        }
        s.pop();
        branch = branch->rchild;
    }
}
return sum;
}
int PreOrderAtIndex(BiTree const &T, int index,
int(*Visit)(TElemType))
{
    if (!T || index < 1)
        return index;

    if (index == 1)
    {
        Visit(T->data);
        return 0;
    }
    index = PreOrderAtIndex(T->lchild, --index, Visit);
    PreOrderAtIndex(T->rchild, index, Visit);
}
void PreOrderAtIndexLegacy(BiTree const & T, int const index,
int(*Visit)(TElemType))
{
    stack<BiTree> s;
    BiTree branch = T;
    int i = 1;
    while (branch || !s.empty())
    {
        if (i == index)
        {
            Visit(branch->data);
            break;
        }
        i++;
        s.push(branch);
        branch = branch->lchild;
        while (!branch && !s.empty())
        {
            branch = s.top();
            s.pop();
            branch = branch->rchild;
        }
    }
}
int main(int argc, char const *argv[])
{

```

<pre>BiTree T; //declaration cout << "Input a tree :" << endl; //用例:ABD..EH...CF.I..G..\n CreateBiTree(T); //创建 Visit = &PrintElement; PreOrderAtIndex(T, 4, Visit); cout << endl; PreOrderAtIndexLegacy(T, 4, Visit); cout << endl << countLeaves(T) << endl; cout << countLeavesLegacy(T) << endl; return 0; }</pre>
<p>五、实验数据记录和处理</p> <pre>Input a tree : ABD..EH...CF.I..G.. E E 4 4 请按任意键继续. . .</pre>
<p>六、讨论、心得</p> <p>二叉树是计算机中必不可少的结构，灵活运用二叉树可以解决很多问题</p>

图结构

一、实验目的和要求

熟悉图的存储结构，掌握有关算法的实现，了解图在计算机科学及其他工程技术中的应用。

二、实验内容和原理

1. 采用邻接表存储结构，编写一个求无向图的连通分量个数的算法。

2. 试基于图的深度优先搜索策略编写一程序，判别以邻接表方式存储的有向图中是否存在有顶点 V_i 到 V_j 顶点的路径($i \neq j$)。

3. 在上述例题中，如改用邻接表的方式存储图，试编一程序实现上述算法。

顶点表 nodelist 的每个元素包含四个字段：

info	mark	pre	out
------	------	-----	-----

其中 mark 为布尔类型，用来标记顶点是否被访问过。开始时，所有元素的 mark 字段为 false，每访问过一个顶点，则 mark 字段置为 true。info 为顶点值，pre 为访问路径上该顶点的前驱顶点的序号，out 指向该顶点的出边表。

三、主要仪器设备

使用的计算机：笔记本或台式计算机，Visual Studio 2015

四、操作方法与实验步骤

2.

```

#include <stdio.h>
#include <stdlib.h>
#include "queue.h"
#define MAX_VERTEX_NUM 20
typedef char VertexType;
typedef struct ArcNode
{
    int adjvex;
    struct ArcNode *nextarc = nullptr;
} ArcNode;

typedef struct VNode
{
    VertexType data;
    ArcNode *firstarc = nullptr;
} VNode, AdjList[MAX_VERTEX_NUM];

typedef struct
{
    AdjList vertices;
    int vexnum, arcnum;
} ALGraph;
bool visited[MAX_VERTEX_NUM];
void CreateGraph(ALGraph *G)
{
    int i, j, k;
    char a;
    ArcNode *s;           //定义边表结点
    printf("Input VertexNum(n) and ArcNum(e): ");
    scanf("%d,%d", &G->vexnum, &G->arcnum);    //读入顶点数和边数
    scanf("%c", &a);
    printf("Input Vertex string:");
    for (i = 0; i < G->vexnum; i++)           //建立边表
    {
        scanf("%c", &a);
        G->vertices[i].data = a;           //读入顶点信息
        G->vertices[i].firstarc = NULL;    //边表置为空表
    }
    printf("Input edges, Creat Adjacency List\n");
    for (k = 0; k < G->arcnum; k++)
    {
        //建立边表
        scanf("%d,%d", &i, &j);           //读入边 (Vi, Vj) 的顶点对序号
        s = (ArcNode *)malloc(sizeof(ArcNode));    //生成边表结点
        s->adjvex = j;           //邻接点序号为j
        s->nextarc = G->vertices[i].firstarc;
        G->vertices[i].firstarc = s;    //将新结点*S插入顶点Vi的边表头部
        s = (ArcNode *)malloc(sizeof(ArcNode));
        s->adjvex = i;           //邻接点序号为i
        s->nextarc = G->vertices[j].firstarc;
    }
}

```

```

        G->vertices[j].firstarc = s;    //将新结点*S插入顶点Vj的边表头
    }
}
void DFSTraverse(ALGraph *G, int i)
{
    //以Vi为出发点对邻接链表表示的图G进行DFS搜索
    ArcNode *p;
    printf("%c", G->vertices[i].data);    //访问顶点Vi
    visited[i] = true;                    //标记Vi已访问
    p = G->vertices[i].firstarc;          //取Vi边表的头指针
    while (p)
    {
        //依次搜索Vi的邻接点Vj，这里j=p->adjvex
        if (!visited[p->adjvex])        //若Vj尚未被访问
            DFSTraverse(G, p->adjvex);    //则以Vj为出发点向纵深搜索
        p = p->nextarc;                  //找Vi的下一个邻接点
    }
}
void DFS(ALGraph *G)
{
    int i;
    for (i = 0; i < G->vexnum; i++)
        visited[i] = false;            //标志向量初始化
    for (i = 0; i < G->vexnum; i++)
        if (!visited[i])                //Vi未访问过
            DFSTraverse(G, i);          //以Vi为源点开始DFS搜索
} //以Vi为源点开始DFS搜索
void BFS(ALGraph *G, int k)
{
    //以Vk为源点对用邻接链表表示的图G进行广度优先搜索
    int i, f = 0, r = 0;
    ArcNode *p;
    int cq[MAX_VERTEX_NUM];            //定义FIFO队列
    for (i = 0; i < G->vexnum; i++)
        visited[i] = false;            //标志向量初始化
    for (i = 0; i <= G->vexnum; i++)
        cq[i] = -1;                    //初始化标志向量
    printf("%c", G->vertices[k].data); //访问源点Vk
    visited[k] = true;
    cq[r] = k;                          //Vk已访问，将其入队。注意，实际上是将其序号入队
    while (cq[f] != -1)
    {
        //队列非空则执行
        i = cq[f]; f = f + 1;           //Vi出队
        p = G->vertices[i].firstarc;    //取Vi的边表头指针
        while (p)
        {
            //依次搜索Vi的邻接点Vj（令p->adjvex=j）
            if (!visited[p->adjvex])
            {
                //若Vj未访问过
                printf("%c", G->vertices[p->adjvex].data); //访问
            }
        }
    }
}

```

```

Vj
        visited[p->adjvex] = true;
        r = r + 1; cq[r] = p->adjvex;           //访问过的Vj入队
    }
    p = p->nextarc;           //找Vi的下一个邻接点
}
} //endwhile
}
bool DfsReachable(ALGraph *g, int i, int j)
{
    ArcNode *p;
    SqQueue Q;
    int e, k;
    InitQueue(Q);
    if (!&g->vexnum || !&g->arcnum) //图的当前顶点数或弧数为0时
        return ERROR;
    else
    {
        EnQueue(Q, i);
        while (!QueueEmpty(Q))
        {
            DeQueue(Q, e);
            visited[e] = true; //标记被访问过的顶点
            p = g->vertices[e].firstarc;
            for (; p != NULL; p = p->nextarc)
            {
                k = p->adjvex; //当前弧所指向顶点的位置
                if (k == j)
                    return OK;
                else if (!visited[k]) //当前顶点未被访问过
                    EnQueue(Q, k);
            }
        }
        return ERROR;
    }
}
int main()
{
    ALGraph *G;
    G = (ALGraph *)malloc(sizeof(ALGraph));
    CreateGraph(G);
    int m, n;
    printf("2 Numbers\n");
    scanf("%d %d", &m, &n);
    m = DfsReachable(G, m, n);
    if (m)
        printf("Reachable\n");
    else
        printf("Unreachable\n");
}

```

```
}
```

五、实验数据记录和处理

```
Input VertexNum(n) and ArcNum(e): 5,3
Input Vertex string:abcde
Input edges,Creat Adjacency List
1,3
1,2
1,1
2 Vertex
1 5
Unreachable
请按任意键继续. . . ■
```

六、讨论、心得

图的型结构可以解决生活中的好多问题，如最短路径和关键路径

查找

一、实验目的和要求

通过本次实验，掌握查找表上的有关查找方法，并分析时间复杂度。

二、实验内容和原理

1. 编写程序实现下面运算：在二叉排序树中查找关键字为 key 的记录。
2. 试将折半查找的算法改写成递归算法。

三、主要仪器设备

使用的计算机：笔记本或台式计算机，Visual Studio 2015

四、操作方法与实验步骤

1.

```
#include<stdio.h>
#include<stdlib.h>
#define TRUE 1
#define FALSE 0
#define OK 1
#define N 10
#define EQ(a,b) ((a)==(b))
#define LT(a,b) ((a)<(b))
typedef int Status;
typedef int Boolean;
typedef int KeyType;
typedef struct
{
    KeyType key;
    int others;
} ElemType;
typedef ElemType TElemType;
typedef struct BiTNode
{
    TElemType data;
    struct BiTNode *lchild, *rchild;
}BiTNode, *BiTree;
Status initDSTable(BiTree *DT)
```



```

{    // 操作结果：构造一个空的动态查找表DT
    *DT = NULL;
    return OK;
}

BiTree searchBST(BiTree T, KeyType key)
{
    if ((!T) || EQ(key, T->data.key))
        return T;
    else if LT(key, T->data.key)           // 在左子树中继续查找
        return searchBST(T->lchild, key);
    else
        return searchBST(T->rchild, key);   // 在右子树中继续查找
}

void searchBST(BiTree *T, KeyType key, BiTree f, BiTree *p, Status
*flag)
{
    if (!*T)
    {
        *p = f;
        *flag = FALSE;
    }
    else if EQ(key, (*T)->data.key)
    {
        *p = *T;
        *flag = TRUE;
    }
    else if LT(key, (*T)->data.key)
        searchBST(&(*T)->lchild, key, *T, p, flag); // 在左子树中继续
查找
    else
        searchBST(&(*T)->rchild, key, *T, p, flag); // 在右子树中继续
查找
    }
}

Status insertBST(BiTree *T, ElemType e)
{
    BiTree p, s;
    Status flag;
    searchBST(T, e.key, NULL, &p, &flag);
    if (!flag)
    {
        s = (BiTree)malloc(sizeof(BiTNode));
        s->data = e;
        s->lchild = s->rchild = NULL;
        if (!p)
            *T = s;
        else if LT(e.key, p->data.key)
            p->lchild = s;
        else
            p->rchild = s;
    }
}

```

```

        return TRUE;
    }
    else
        return FALSE;
}
void deleteBST(BiTree *p)
{
    BiTree q, s;
    if (!(*p)->rchild)
    {
        q = *p;
        *p = (*p)->lchild;
        free(q);
    }
    else if (!(*p)->lchild)
    {
        q = *p;
        *p = (*p)->rchild;
        free(q);
    }
    else
    {
        q = *p;
        s = (*p)->lchild;
        while (s->rchild)
        {
            q = s;
            s = s->rchild;
        }
        (*p)->data = s->data;
        if (q != *p)
            q->rchild = s->lchild;
        else
            q->lchild = s->lchild;
        free(s);
    }
}
void traverseDSTable(BiTree DT, void(*Visit)(ElemType))
{
    if (DT)
    {
        traverseDSTable(DT->lchild, Visit);
        Visit(DT->data);
        traverseDSTable(DT->rchild, Visit); // 最后中序遍历右子树
    }
}
void print(ElemType c)
{
    printf("(%d,%d) ", c.key, c.others);
}

```

```

}
void main()
{
    BiTree dt, p;
    int i;
    KeyType j;
    ElemType r[N] = { { 45, 1 }, { 12, 2 }, { 53, 3 }, { 3, 4 }, { 37, 5 }, { 24,
6 }, { 100, 7 }, { 61, 8 }, { 90, 9 }, { 78, 10 } };
    initDSTable(&dt);
    for (i = 0; i < N; i++)
        insertBST(&dt, r[i]);
    traverseDSTable(dt, print);
    printf("\nPlease input the value you want to find: ");
    scanf_s("%d", &j);
    p = searchBST(dt, j);
    if (p)
    {
        printf("Value has been find in this tree\n.");
    }
    else
        printf("Can not find value in this tree\n");
}

```

```

2.
#include <iostream>
using namespace std;
int binarySearch(int *arr, int first, int end, int key)
{
    int f, mid;
    mid = (first + end) / 2;
    if (first > end)
        f = -1;
    else if (arr[mid] == key)
        f = mid;
    else if (arr[mid] > key)
        f = binarySearch(arr, first, end - 1, key);
    else
        f = binarySearch(arr, first + 1, end, key);
    return f;
}
int main()
{
    int a[10] = { 11, 13, 18, 22, 35, 44, 56, 79, 82, 103 };
    int p, k;
    cout << "input k:";
    cin >> k;
    p = binarySearch(a, 0, 6, k);
    if (p >= 0)
        cout << "found:" << k << "=a[" << p + 1 << "]" << endl;
    else

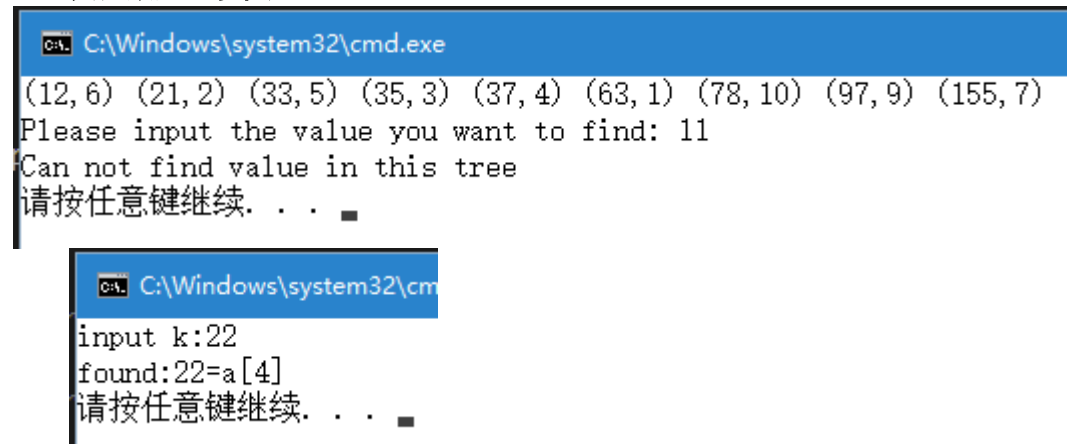
```

```

        cout << "no found:" << k << endl;
        return 0;
    }

```

五、实验数据记录和处理



```

C:\Windows\system32\cmd.exe
(12,6) (21,2) (33,5) (35,3) (37,4) (63,1) (78,10) (97,9) (155,7)
Please input the value you want to find: 11
Can not find value in this tree
请按任意键继续. . .

C:\Windows\system32\cmd.exe
input k:22
found:22=a[4]
请按任意键继续. . .

```

六、讨论、心得

搜索在当代信息繁杂的时代是不可缺少的，一个好的算法可以使信息来的更加简单

排序

一、实验目的和要求

通过本次实验，掌握线性表的排序方法，并分析时间复杂度。

二、实验内容和原理

设计一个用链表表示的直接选择排序算法，并用程序实现。

三、主要仪器设备

使用的计算机：笔记本或台式计算机，Visual Studio 2015

四、操作方法与实验步骤

```

#include <iostream>
using namespace std;
typedef struct LNode { //单链表节点结构
    int data;
    struct LNode *next;
}LNode, *LinkList;
int printList(LinkList &L) { //输出单链表
    int i = 1;
    LinkList p = L;

    while (p) {
        cout << p->data<<"\t";
        p = p->next;
        i++;
    }
    cout << endl;
    return 1;
} //printList
int createList(LinkList &L, int data[], int n) { //创建带头单链表,P30
    L = (LinkList)malloc(sizeof(LNode));

```

```

    L->data = 0;
    L->next = NULL;
    LinkList p;
    for (int i = 0; i < n; i++) {
        p = (LinkList)malloc(sizeof(LNode));
        p->data = data[i];
        p->next = L->next;
        L->next = p;
    }
    return 1;
}

void sort(LinkList head, int n) {
    if (n == 0)
        return;
    int min;
    int i, s = 0;
    LinkList p;
    LinkList r;
    LinkList t;
    p = head->next;
    r = head->next;
    t = head;
    min = p->data;
    for (i = 0; i < n; i++) {
        if (min > p->data) {
            min = p->data;
            r = p;
            s = i;
        }
        p = p->next;
    }
    for (i = 0; i < s; i++)
        t = t->next;
    t->next = r->next;
    r->next = head->next;
    head->next = r;
    sort(head->next, n - 1);
}

int main()
{
    int data[] = {17, 35, 29, 68, 93, 77};
    LinkList L;
    createList(L, data, 6);
    cout << "原始链表: " << endl;
    printList(L); // 输出
    cout << endl;
    sort(L, 5);
    printList(L);
    return 0;
}

```

}

五、实验数据记录和处理

```
C:\Windows\system32\cmd.exe
原始链表:
0      77      93      68      29      35      17
0      29      35      68      77      93      17
请按任意键继续. . .
```

六、讨论、心得

在数据堆积如山的时代，排序算法可以大大加强数据的处理效率