

中国地质大学（北京）

课程设计报告

课程名称：数据结构课程设计

学 号：1005183121

姓 名：周子杰

完成时间：2020.9.10

目录

实验一 线性表实验.....	2
实验内容题目一顺序表.....	2
实验内容题目二：单链表.....	6
题目四：循环链表.....	12
实验二 栈、队列实验.....	19
实验内容题目一：栈.....	19
实验内容题目二：循环队列.....	22
实训项目题目五：模拟看病.....	26
实验三 串和数组实验.....	31
实验内容题目二：KMP.....	31
实验内容题目三：稀疏矩阵.....	33
实验内容题目五：马鞍点.....	38
实验四 树实验.....	41
实验内容题目一：二叉树.....	41
实验内容题目二：前序中序构造二叉树.....	50
实验内容题目四：线索化二叉树.....	58
实验五 图实验.....	68
实验内容题目一：邻接矩阵-邻接表.....	68
实验内容题目二：图的遍历.....	72
实训项目题目一：子工程建设时间的安排.....	77
实验六 查找表实验.....	81
实验内容题目一：顺序查找.....	81
实验内容题目二：二分查找.....	82
实验内容题目三：哈希查找.....	83
实验七 内排序实验.....	86
实验内容题目一：直接插入.....	86
实验内容题目二：希尔排序.....	87
实验内容题目三：快速排序.....	89

实验一 线性表实验

【实验目的】

1. 掌握顺序表、单链表、循环链表、双向链表的构造原理及其基本运算的实现算法。
2. 了解线性表的顺序存储和链式存储结构的特点和适用情形。

【实验学时】

6 学时

【实验内容】

实验内容题目一顺序表

题目一：编写一个程序，实现顺序表的各种基本运算，并在此基础上设计一个主程序完成如下功能：

- (1) 初始化顺序表 L。
- (2) 依次采用尾插法插入 a, b, c, d, e 元素。
- (3) 输出顺序表 L 及 L 的长度。
- (4) 判断顺序表 L 是否为空。
- (5) 输出顺序表 L 的第 3 个元素。
- (6) 输出元素 d 的位置。
- (7) 在第 4 个元素位置上插入 f 元素。
- (8) 删除 L 的第 3 个元素。
- (9) 输出顺序表 L。
- (10) 释放顺序表 L。

代码

seqlist.h

```

1. #ifndef SEQLIST_H
2. #define SEQLIST_H
3.
4. #include <iostream>
5.
6. class seqList
7. {
8. private:
9.     char *data = new char[100]; //存放元素的数组
10.    int last=-1;    //最后一个元素位置(从 0 开始)
11.
12. public:
13.    seqList(){} //构建
14.    ~seqList(){delete [] data;} //析构(释放)

```

```

15. void insertData(char d); //尾插法插入元素
16. int listLength(); //获取长度
17. bool isEmpty(); //判断是否为空
18. char getDataByNum(int n); //获取第 n 个元素
19. int getNumByData(char d); //获取元素 d 的位置
20. void insertDataByNum(int n, char d); //在第 n 个元素的位置上插入 d
21. void deleteDataByNum(int n); //删除第 n 个元素
22. void outputList(); //输出整个表
23. };
24.
25. #endif // SEQLIST_H

```

seqlist.cpp

```

1. #include "seqlist.h"
2.
3. void seqList::insertData(char d){
4.     *(data+last+1)=d;
5.     last++;
6. }
7.
8. int seqList::listLength(){
9.     return last+1;
10. }
11.
12. bool seqList::isEmpty(){
13.     return last == -1;
14. }
15.
16. char seqList::getDataByNum(int n){
17.     if(n>last+1||n<=0)
18.         return 0; //表示未找到
19.     return *(data+n-1);
20. }
21.
22. int seqList::getNumByData(char d){
23.     for(int i=0; i<=last; ++i)
24.         if(*(data+i) == d)
25.             return i+1;
26.     return 0; //表示未找到
27. }
28.
29. void seqList::insertDataByNum(int n, char d){
30.     if(n>=last+1||n<=0)

```

```

31.     printf("请输入合适的位置\n");
32.     else{
33.         for(int i=last; i>=n-1; --i)
34.             *(data+i+1)=*(data+i);
35.         data[n-1]=d;
36.         last++;
37.     }
38. }
39.
40. void seqList::deleteDataByNum(int n){
41.     if(n>=last+1 || n<=0)
42.         printf("请输入合适的位置\n");
43.     else{
44.         for(int i=n-1; i<=last; ++i)
45.             *(data+i)=*(data+i+1);
46.         last--;
47.     }
48. }
49.
50. void seqList::outputList(){
51.     for(int i=0; i<=last; ++i)
52.         printf("%c ",*(data+i));
53. }

```

main.cpp

```

1. #include <iostream>
2. #include "seqlist.h"
3.
4. int main()
5. {
6.     //初始化顺序表 L
7.     seqList L;
8.
9.     //尾插法插入
10.    L.insertData('a');
11.    L.insertData('b');
12.    L.insertData('c');
13.    L.insertData('d');
14.    L.insertData('e');
15.
16.    //输出长度
17.    printf("顺序表的长度为: %d\n",L.listLength());
18.

```

```

19.    //判断为空
20.    bool isempty = L.isEmpty();
21.    if(isempty)
22.        printf("顺序表为空\n");
23.    else
24.        printf("顺序表不为空\n");
25.
26.    //输出第三个元素
27.    char d3 = L.getDataByNum(3);
28.    if(d3 == 0)
29.        printf("请选择正确的位置\n");
30.    else
31.        printf("第三个元素为: %c\n",d3);
32.
33.    //输出元素 d 的位置
34.    int nd = L.getNumByData('d');
35.    if(nd == 0)
36.        printf("未找到该元素\n");
37.    else
38.        printf("该元素所在位置为: %d\n",nd);
39.
40.    //在第四个元素位置上插入 f 元素
41.    L.insertDataByNum(4, 'f');
42.
43.    //删除第三个元素
44.    L.deleteDataByNum(3);
45.
46.    //输出全部
47.    L.outputList();
48.
49.    //释放
50.    L.~seqList();
51. }

```

运行截图:

```

F:\QT\setup\Tools\QtCreator\bin\qtcreator_process_stub.exe
顺序表的长度为: 5
顺序表不为空
第三个元素为: c
该元素所在位置为: 4
a b f d e

```

实验内容题目二：单链表

题目二：编写一个程序，实现单链表的各种基本运算，并在此基础上设计一个主程序完成如下功能：

- (1) 初始化单链表 H。
- (2) 依次采用尾插法插入 a, b, c, d, e 元素。
- (3) 输出单链表 H 及 H 的长度。
- (4) 判断单链表 H 是否为空。
- (5) 输出单链表 H 的第 3 个元素。
- (6) 输出元素 d 的位置。
- (7) 在第 4 个元素位置上插入 f 元素。
- (8) 删除 H 的第 3 个元素。
- (9) 输出单链表 L。
- (10) 释放单链表 L。

代码

list.h

```

1. #ifndef LIST_H
2. #define LIST_H
3.
4. #include <iostream>
5.
6. //采用结构方式
7. //链表结点类
8. struct LinkNode{
9.     char data;

```

```

10.     LinkNode * next;
11.     LinkNode(LinkNode * p = NULL){next = p;}
12.     LinkNode(char &d, LinkNode * p = NULL){
13.         data = d;
14.         next = p;
15.     }
16. };
17.
18. //链表类
19. class List
20. {
21. private:
22.     LinkNode * head;    //头结点
23.     LinkNode * last;    //指向链尾 - 方便尾插法
24.
25. public:
26.     List(){head = new LinkNode;} //构造
27.     ~List(){makeEmpty();} //析构（释放）
28.     void makeEmpty(); //置为空表
29.     void insertData(char &d); //尾插法插入元素
30.     LinkNode * getHead(){return head;} //获取头结点
31.     int listLength(); //获取长度
32.     bool isEmpty(); //判断是否为空
33.     char getDataByNum(int n); //获取第 n 个元素
34.     int getNumByData(char &d); //获取元素 d 的位置
35.     LinkNode * locate(int n); //返回第 n 个元素地址
36.     bool insertDataByNum(int n, char &d); //在第 n 个元素的位置上插入 d
37.     bool deleteDataByNum(int n); //删除第 n 个元素
38.     void outputList(); //输出整个表
39.     List & operator = (List & L);
40. };
41.
42. #endif // LIST_H

```

list.cpp

```

1. #include "list.h"
2.
3. void List::makeEmpty(){
4.     LinkNode *p;
5.     while(head->next != NULL){
6.         p = head->next;
7.         head->next = p->next;
8.         delete p;

```



```

9.     }
10. }
11.
12. void List::insertData(char &d){
13.     LinkNode * newNode; //要插入的新结点
14.     if(isEmpty()){
15.         head = new LinkNode;
16.         last = head;
17.     }
18.     newNode = new LinkNode(d);
19.     last->next = newNode;
20.     last = newNode;
21. }
22.
23. int List::listLength(){
24.     LinkNode *p = head->next;
25.     int i = 0;
26.     while(p != NULL){
27.         p = p->next;
28.         i++;
29.     }
30.     return i;
31. }
32.
33. bool List::isEmpty(){
34.     return head->next == NULL;
35. }
36.
37. char List::getDataByNum(int n){
38.     if(n<=0||n>listLength())
39.         return 0; //未找到
40.     LinkNode *p = head;
41.     while(n--){
42.         p = p->next;
43.     }
44.     return p->data;
45. }
46.
47. int List::getNumByData(char &d){
48.     LinkNode *p = head->next;
49.     int i = 0;
50.     while(p != NULL){
51.         if(p->data == d) break;
52.         else p = p->next;

```

```

53.         i++;
54.     }
55.     if(i>listLength())
56.         return 0;    //未找到
57.     return i;
58. }
59.
60. LinkNode * List::locate(int n){
61.     if(n<0||n>listLength())
62.         return NULL;
63.     LinkNode *p = head;
64.     for(int i=0; i<n; ++i){
65.         p = p->next;
66.     }
67.     return p;
68. }
69.
70. bool List::insertDataByNum(int n, char &d){
71.     // if(n<=0||n>listLength()+1)
72.     //     printf("请输入合适的位置");
73.     LinkNode *p = locate(n);
74.     if(p == NULL){
75.         //printf("插入失败");
76.         return false;
77.     }
78.     LinkNode *newNode = new LinkNode(d);
79.     newNode->next = p->next;
80.     p->next = newNode;
81.     //printf("插入成功");
82.     return true;
83. }
84.
85. bool List::deleteDataByNum(int n){
86.     if(n<=0||n>listLength()){
87.         //printf("请输入合适的位置");
88.         return false;
89.     }
90.     LinkNode *p = locate(n-1);
91.     LinkNode *del = p->next;
92.     p->next = del->next;
93.     delete del;
94.     //printf("删除成功");
95.     return true;
96. }

```

```

97.
98. void List::outputList(){
99.     LinkNode *p = head;
100.     for(int i=0; i<listLength(); ++i){
101.         printf("%c ",p->next->data);
102.         p = p->next;
103.     }
104. }
105.
106. List& List::operator=(List & L){
107.     char value;
108.     LinkNode * s = L.getHead();
109.     LinkNode * d = head = new LinkNode;
110.     while(s->next != NULL){
111.         value = s->next->data;
112.         d->next = new LinkNode(value);
113.         d = d->next;
114.         s = s->next;
115.     }
116.     d->next = NULL;
117.     return * this;
118. }

```

main.cpp

```

1. #include <iostream>
2. #include "list.h"
3.
4. int main()
5. {
6.     //初始化单链表 H
7.     List H;
8.
9.     //尾插法插入
10.    char a='a',b='b',c='c',d='d',e='e',f='f';
11.    H.insertData(a);
12.    H.insertData(b);
13.    H.insertData(c);
14.    H.insertData(d);
15.    H.insertData(e);
16.
17.    //输出长度
18.    printf("单链表的长度为: %d\n",H.listLength());
19.

```

```

20.    //判断为空
21.    bool isempty = H.isEmpty();
22.    if(isempty)
23.        printf("单链表为空\n");
24.    else
25.        printf("单链表不为空\n");
26.
27.    //输出第三个元素
28.    char d3 = H.getDataByNum(3);
29.    if(d3 == 0)
30.        printf("请选择正确的位置\n");
31.    else
32.        printf("第三个元素为: %c\n",d3);
33.
34.    //输出元素 d 的位置
35.    int nd = H.getNumByData(d);
36.    if(nd == 0)
37.        printf("未找到该元素\n");
38.    else
39.        printf("该元素所在位置为: %d\n",nd);
40.
41.    //在第四个元素位置上插入 f 元素
42.    if(H.insertDataByNum(4,f))
43.        printf("插入成功\n");
44.    else
45.        printf("插入失败\n");
46.
47.    //删除第三个元素
48.    if(H.deleteDataByNum(3))
49.        printf("删除成功\n");
50.    else
51.        printf("删除失败\n");
52.
53.    //输出全部
54.    H.outputList();
55.
56.    //释放
57.    H.~List();
58. }

```

运行截图

```

F:\QT\setup\Tools\QtCreator\bin\qtcreator_process_stub.exe
单链表的长度为: 5
单链表不为空
第三个元素为: c
该元素所在位置为: 3
插入成功
删除成功
a b d f e
    
```

题目四: 循环链表

题目四: 编写一个程序, 实现循环单链表的各种基本运算, 并在此基础上设计一个主程序完成如下功能:

- (1) 初始化循环单链表 H。
- (2) 依次采用尾插法插入 a, b, c, d, e 元素。
- (3) 输出循环单链表 H 及 H 的长度。
- (4) 判断循环单链表 H 是否为空。
- (5) 输出循环单链表 H 的第 3 个元素。
- (6) 输出元素 d 的位置。
- (7) 在第 4 个元素位置上插入 f 元素。
- (8) 删除 H 的第 3 个元素。
- (9) 输出循环单链表 L。
- (10) 释放循环单链表 L。

代码

list.h

```

1. #ifndef CIRCLINK_H
2. #define CIRCLINK_H
3.
4. #include <iostream>
5.
6. struct circNode{
7.     char data;
8.     circNode * next;
9.     circNode(circNode *link = NULL){
    
```

```

10.         next = link;
11.     }
12.     circNode(char d, circNode *link = NULL){
13.         data = d;
14.         next = link;
15.     }
16. };
17.
18. class List
19. {
20. private:
21.     circNode *head, *last;
22.
23. public:
24.     List(){head = new circNode; head->next = head;}
25.     ~List(){makeEmpty();}    //析构（释放）
26.     void makeEmpty();    //置为空表
27.     void insertData(char &d); //尾插法插入元素
28.     circNode * getHead(){return head;} //获取头结点
29.     int listLength();    //获取长度
30.     bool isEmpty(); //判断是否为空
31.     char getDataByNum(int n); //获取第 n 个元素
32.     int getNumByData(char &d);    //获取元素 d 的位置
33.     circNode * locate(int n);    //返回第 n 个元素地址
34.     bool insertDataByNum(int n, char &d);    //在第 n 个元素的位置上插入 d
35.     bool deleteDataByNum(int n);    //删除第 n 个元素
36.     void outputList(); //输出整个表
37.     List & operator = (List & L);
38. };
39.
40. #endif // CIRCLINK_H

```

list.cpp

```

1. #include "list.h"
2.
3. void List::makeEmpty(){
4.     head->next = head;
5. }
6.
7. void List::insertData(char &d){
8.     circNode * newNode;
9.
10.     if(isEmpty()){

```

```

11.         head = new circNode;
12.         last = head;
13.     }
14.     newNode = new circNode(d);
15.     last->next = newNode;
16.     last = newNode;
17.     last->next = head;
18. }
19.
20. int List::listLength(){
21.     circNode *p = head->next;
22.     int i = 0;
23.     while(p->next != head){
24.         p = p->next;
25.         i++;
26.     }
27.     return i;
28. }
29.
30. bool List::isEmpty(){
31.     return head->next == head;
32. }
33.
34. char List::getDataByNum(int n){
35.     if(n<=0||n>listLength())
36.         return 0;    //未找到
37.     circNode *p = head;
38.     while(n--){
39.         p = p->next;
40.     }
41.     return p->data;
42. }
43.
44. int List::getNumByData(char &d){
45.     circNode *p = head->next;
46.     int i = 0;
47.     while(p->next != head){
48.         if(p->data == d) break;
49.         else p = p->next;
50.         i++;
51.     }
52.     if(i>listLength())
53.         return 0;    //未找到
54.     return i;

```

```

55. }
56.
57. circNode * List::locate(int n){
58.     if(n<0||n>listLength())
59.         return NULL;
60.     circNode *p = head;
61.     for(int i=0; i<n; ++i){
62.         p = p->next;
63.     }
64.     return p;
65. }
66.
67. bool List::insertDataByNum(int n, char &d){
68.     // if(n<=0||n>listLength()+1)
69.     //     printf("请输入合适的位置");
70.     circNode *p = locate(n);
71.     if(p == NULL){
72.         //printf("插入失败");
73.         return false;
74.     }
75.     circNode *newNode = new circNode(d);
76.     newNode->next = p->next;
77.     p->next = newNode;
78.     if(n == listLength()){
79.         last = newNode;
80.         last->next = head;
81.     }
82.     //printf("插入成功");
83.     return true;
84. }
85.
86. bool List::deleteDataByNum(int n){
87.     if(n<=0||n>listLength()){
88.         //printf("请输入合适的位置");
89.         return false;
90.     }
91.     circNode *p = locate(n-1);
92.     circNode *del = p->next;
93.
94.     if(n == listLength()){
95.         last = p;
96.         last->next = head;
97.     }
98.

```



```

99.     p->next = del->next;
100.     delete del;
101.     //printf("删除成功");
102.     return true;
103. }
104.
105. void List::outputList(){
106.     circNode *p = head;
107.     for(int i=0; i<=listLength(); ++i){
108.         printf("%c ",p->next->data);
109.         p = p->next;
110.     }
111. }
112.
113. List& List::operator=(List & L){
114.     char value;
115.     circNode * s = L.getHead();
116.     circNode * d = head = new circNode;
117.     while(s->next != NULL){
118.         value = s->next->data;
119.         d->next = new circNode(value);
120.         d = d->next;
121.         s = s->next;
122.     }
123.     d->next = NULL;
124.     return * this;
125. }

```

main.cpp

```

1. #include <iostream>
2. #include "list.h"
3.
4. int main()
5. {
6.     //初始化循环链表 H
7.     List H;
8.
9.     //尾插法插入
10.    char a='a',b='b',c='c',d='d',e='e',f='f';
11.    H.insertData(a);
12.    H.insertData(b);
13.    H.insertData(c);
14.    H.insertData(d);

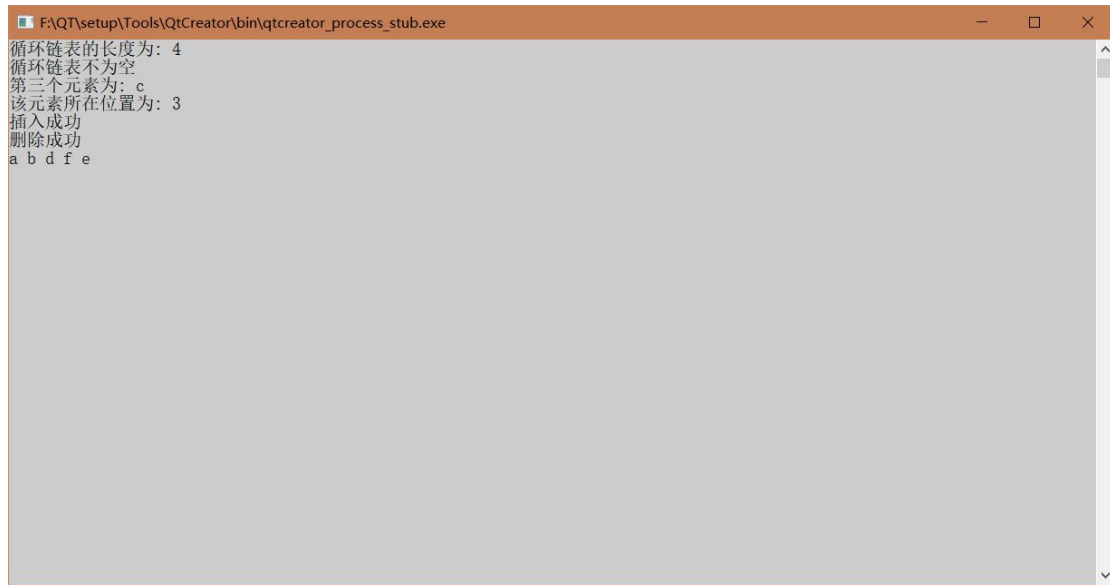
```

```

15.     H.insertData(e);
16.
17.     //输出长度
18.     printf("循环链表的长度为: %d\n",H.listLength());
19.
20.     //判断为空
21.     bool isempty = H.isEmpty();
22.     if(isempty)
23.         printf("循环链表为空\n");
24.     else
25.         printf("循环链表不为空\n");
26.
27.     //输出第三个元素
28.     char d3 = H.getDataByNum(3);
29.     if(d3 == 0)
30.         printf("请选择正确的位置\n");
31.     else
32.         printf("第三个元素为: %c\n",d3);
33.
34.     //输出元素 d 的位置
35.     int nd = H.getNumByData(d);
36.     if(nd == 0)
37.         printf("未找到该元素\n");
38.     else
39.         printf("该元素所在位置为: %d\n",nd);
40.
41.     //在第四个元素位置上插入 f 元素
42.     if(H.insertDataByNum(4,f))
43.         printf("插入成功\n");
44.     else
45.         printf("插入失败\n");
46.
47.     //删除第三个元素
48.     if(H.deleteDataByNum(3))
49.         printf("删除成功\n");
50.     else
51.         printf("删除失败\n");
52.
53.     //输出全部
54.     H.outputList();
55.
56.     //释放
57.     H.~List();
58. }

```

运行截图



```
F:\QT\setup\Tools\QtCreator\bin\qtcreator_process_stub.exe
循环链表的长度为: 4
循环链表不为空
第三个元素为: c
该元素所在位置为: 3
插入成功
删除成功
a b d f e
```

实验二 栈、队列实验

【实验目的】

1. 掌握栈的顺序及链式存储和基本运算的实现方法
2. 掌握队列的顺序及链式存储和基本运算的实现方法
3. 了解栈和队列的应用

【实验学时】

8 学时

【实验内容】

实验内容题目一：栈

题目一：编写一个程序，实现顺序栈的各种基本运算，并在此基础上设计一个主程序完成如下功能：

- (1) 初始化栈 S。
- (2) 判断栈 S 是否非空。
- (3) 依次进栈元素 a, b, c, d, e 。
- (4) 输出栈的长度。
- (5) 输出从栈顶到栈底的元素。
- (6) 输出出栈序列。
- (7) 释放栈。

代码

stack.h

```

1. #ifndef STACK_H
2. #define STACK_H
3.
4. #include <iostream>
5.
6. class Stack
7. {
8. private:
9.     char *data;
10.    int maxSize;    //栈最大容量
11.    int top;        //栈中元素数量 指向栈顶
12.    void overflow();
13.
14. public:
15.    Stack();        //构建
16.    ~Stack(){delete [] data;}    //析构
    
```

```

17. void push(char d); //入栈
18. void pop(); //出栈
19. char getpop(); //获取栈顶元素
20. bool isEmpty(); //判断栈空
21. bool isFull(); //判断栈满
22. int getSize(); //获取栈长
23. void makeEmpty(); //清空栈
24. void outputStack(); //输出栈(从栈顶到栈底)
25. };
26.
27. #endif // STACK_H

```

stack.cpp

```

1. #include "stack.h"
2.
3. Stack::Stack(){
4.     top = -1;
5.     maxSize = 50;
6.     data = new char[maxSize];
7. }
8.
9. void Stack::overflow(){
10.    char * newStack = new char[maxSize+50];
11.    for(int i=0; i<=top; ++i){
12.        newStack[i] = data[i];
13.    }
14.    delete [] data;
15.    data = newStack;
16. }
17.
18. bool Stack::isFull(){
19.    return top == maxSize-1;
20. }
21.
22. bool Stack::isEmpty(){
23.    return top == -1;
24. }
25.
26. void Stack::push(char d){
27.    if(isFull())
28.        overflow();
29.    data[++top] = d;
30.    printf("%c 入栈成功\n",d);

```

```

31. }
32.
33. void Stack::pop(){
34.     if(isEmpty()){
35.         printf("栈为空\n");
36.     }else{
37.         top--;
38.         printf("%c 出栈成功\n",data[top+1]);
39.     }
40. }
41.
42. int Stack::getSize(){
43.     return top+1;
44. }
45.
46. void Stack::makeEmpty(){
47.     top = -1;
48. }
49.
50. void Stack::outputStack(){
51.     for(int i=top; i>=0; --i){
52.         printf("%c ",data[i]);
53.     }
54. }

```

main.cpp

```

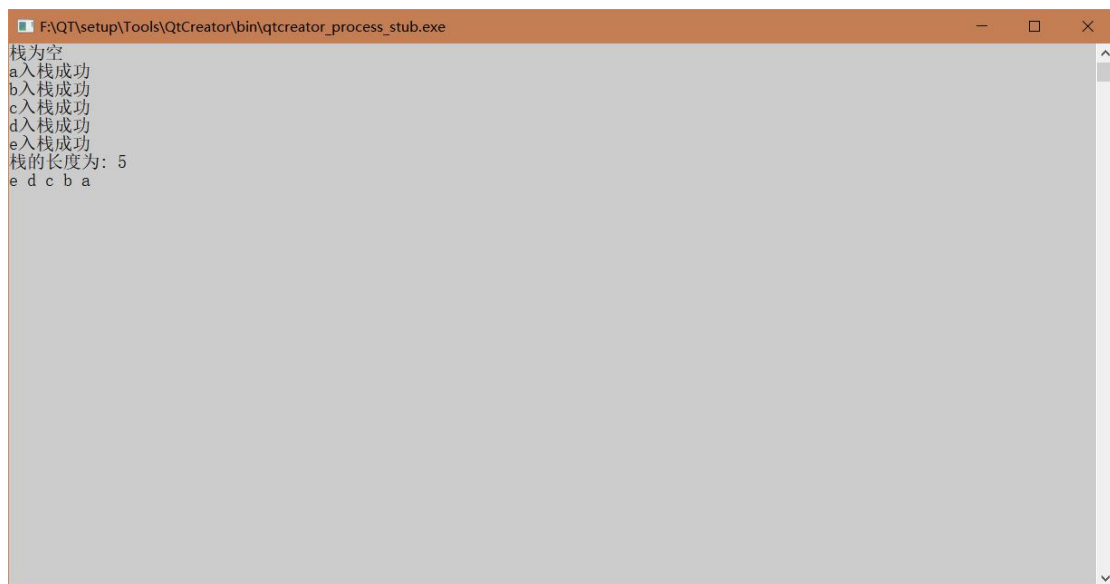
1. #include <iostream>
2. #include "stack.h"
3.
4. int main()
5. {
6.     //初始化栈 S
7.     Stack S;
8.
9.     //判断栈 S 是否非空
10.    if(S.isEmpty()){
11.        printf("栈为空\n");
12.    }else{
13.        printf("栈非空\n");
14.    }
15.
16.    //依次进栈
17.    char a='a',b='b',c='c',d='d',e='e';

```

```

18.    S.push(a);
19.    S.push(b);
20.    S.push(c);
21.    S.push(d);
22.    S.push(e);
23.
24.    //输出栈的长度
25.    printf("栈的长度为: %d\n",S.getSize());
26.
27.    //输出从栈顶到栈底的元素,同时也是出栈序列
28.    S.outputStack();
29.
30.    //释放栈
31.    S.makeEmpty();
32.    S.~Stack();
33. }
    
```

运行截图



实验内容题目二：循环队列

题目二：编写一个程序，实现循环队列的各种基本运算，并在此基础上设计一个主程序完成如下功能：

- (1) 初始化队列 Q。
- (2) 判断队列 Q 是否非空。
- (3) 依次进队列元素 a, b, c。
- (4) 出队一个元素，输出该元素。

- (5) 输出队列 Q 的元素个数。
- (6) 依次进入队列元素 d, e, f。
- (7) 输出出队序列。
- (8) 释放队列。

代码 queue.h

```

1. #ifndef QUEUE_H
2. #define QUEUE_H
3.
4. #include <iostream>
5.
6. //循环队列
7. class Queue
8. {
9. private:
10.     int front, rear;    //队头、队尾
11.     char * data;       //队列
12.     int maxSize;       //最大容量
13.
14. public:
15.     Queue();           //构造
16.     ~Queue(){delete [] data;} //析构
17.     void enqueue(char d); //将 d 入队
18.     void dequeue(); //出队
19.     bool isEmpty(); //判断队空
20.     bool isFull(); //判断队满
21.     int getSize(); //获取队长
22.     void outputQueue(); //输出栈
23. };
24.
25. #endif // QUEUE_H
    
```

queue.cpp

```

1. #include "queue.h"
2.
3. Queue::Queue(){
4.     maxSize = 50;
5.     front = 0;
6.     rear = 0;
7.     data = new char[maxSize];
8. }
    
```



```

9.
10. void Queue::enqueue(char d){
11.     if(isFull()){
12.         printf("队列已满\n");
13.     }else{
14.         data[rear] = d;
15.         rear = (rear+1)%maxSize;
16.         printf("%c 入队成功\n",d);
17.     }
18. }
19.
20. void Queue::dequeue(){
21.     if(isEmpty()){
22.         printf("队列为空\n");
23.     }else{
24.         printf("%c 出队成功\n",data[front]);
25.         front = (front+1)%maxSize;
26.     }
27. }
28.
29. bool Queue::isFull(){
30.     return (rear+1)%maxSize == front;
31. }
32.
33. bool Queue::isEmpty(){
34.     return rear ==front;
35. }
36.
37. int Queue::getSize(){
38.     return (rear-front+maxSize)%maxSize;
39. }
40.
41. void Queue::outputQueue(){
42.     for(int i=front; i!=rear; ++i){
43.         i = i%maxSize;
44.         printf("%c ",data[i]);
45.     }
46. }

```

main.cpp

```

1. #include <iostream>
2. #include "queue.h"
3.

```

```

4. int main()
5. {
6.     //初始化栈 Q
7.     Queue Q;
8.
9.     //判断栈 Q 是否非空
10.    if(Q.isEmpty()){
11.        printf("队列为空\n");
12.    }else{
13.        printf("队列非空\n");
14.    }
15.
16.    //依次入队
17.    char a='a',b='b',c='c';
18.    Q.enqueue(a);
19.    Q.enqueue(b);
20.    Q.enqueue(c);
21.
22.    //出队一个元素并输出该元素
23.    Q.dequeue();
24.
25.    //输出元素个数
26.    printf("队列元素个数为: %d\n",Q.getSize());
27.
28.    //依次入队
29.    char d='d',e='e',f='f';
30.    Q.enqueue(d);
31.    Q.enqueue(e);
32.    Q.enqueue(f);
33.
34.    //输出队列
35.    Q.outputQueue();
36.
37.    //释放队列
38.    Q.~Queue();
39.
40. }

```

运行截图

```

F:\QT\setup\Tools\QtCreator\bin\qtcreator_process_stub.exe
队列为空
a入队成功
b入队成功
c入队成功
a出队成功
队列元素个数为: 2
d入队成功
e入队成功
f入队成功
b c d e f

```

【实训项目】:

实训项目题目五：模拟看病

1. 病人看病模拟程序

编写一个程序，反映病人到医院看病，排队看医生的情况。在病人排队过程中，主要重复两件事：

- (1) 病人到达诊室，将病历本交给护士，排到等待队列中候诊
- (2) 护士从等待队列中取出下一位病人的病历，该病人进入诊室就诊

要求模拟病人等待就诊这一过程。程序采用菜单方式，其选项及功能说明如下：

- 排队——输入排队病人的病历号，加入病人排队队列中；
- 就诊——病人排队队列中最前面的病人就诊，并将其从队列中删除；
- 查看排队——从对首到队尾列出所有的排队病人的病历号；
- 不再排队，余下依次就诊——从对首到队尾列出所有的排队病人的病历号，并退出运行；
- 下班——退出运行。

代码

queue.h

```

1. #ifndef QUEUE_H
2. #define QUEUE_H
3.
4. #include <iostream>
5.
6. //循环队列
7. class Queue
8. {

```

```

9. private:
10.     int front, rear;    //队头、队尾
11.     int * data;        //队列
12.     int maxSize;       //最大容量
13.
14. public:
15.     Queue();           //构造
16.     ~Queue(){delete [] data;} //析构
17.     void enqueue(int d); //将 d 入队
18.     void dequeue(); //出队
19.     bool isEmpty(); //判断队空
20.     bool isFull(); //判断队满
21.     int getSize(); //获取队长
22.     void outputQueue(); //输出栈
23. };
24.
25. #endif // QUEUE_H

```

queue.cpp

```

1. #include "queue.h"
2.
3. Queue::Queue(){
4.     maxSize = 50;
5.     front = 0;
6.     rear = 0;
7.     data = new int[maxSize];
8. }
9.
10. void Queue::enqueue(int d){
11.     if(isFull()){
12.         printf("队列已满\n");
13.     }else{
14.         data[rear] = d;
15.         rear = (rear+1)%maxSize;
16.         printf("%d 号病人入队成功\n",d);
17.     }
18. }
19.
20. void Queue::dequeue(){
21.     if(isEmpty()){
22.         printf("队列为空\n");
23.     }else{
24.         printf("%d 号病人出队成功\n",data[front]);

```

```

25.         front = (front+1)%maxSize;
26.     }
27. }
28.
29. bool Queue::isFull(){
30.     return (rear+1)%maxSize == front;
31. }
32.
33. bool Queue::isEmpty(){
34.     return rear ==front;
35. }
36.
37. int Queue::getSize(){
38.     return (rear-front+maxSize)%maxSize;
39. }
40.
41. void Queue::outputQueue(){
42.     for(int i=front; i!=rear; ++i){
43.         i = i%maxSize;
44.         printf("%d ",data[i]);
45.     }
46. }

```

main.cpp

```

1. #include <iostream>
2. #include "queue.h"
3. using namespace std;
4.
5. void menu(Queue Q){
6.     cout<<"菜单:"<<endl
7.         <<"1.排队"<<endl
8.         <<"2.就诊"<<endl
9.         <<"3.查看排队"<<endl
10.        <<"4.不再排队, 余下依次就诊"<<endl
11.        <<"5.下班"<<endl;
12.     while(true){
13.         int choice;
14.         cout<<"请输入要选择的操作: ";
15.         cin>>choice;
16.         switch (choice) {
17.             case 1:{
18.                 printf("请输入病历号: ");
19.                 int blh;    //病历号

```

```

20.         cin>>blh;
21.         Q.enqueue(blh);
22.         break;
23.     }
24.     case 2:{
25.         Q.dequeue();
26.         break;
27.     }
28.     case 3:{
29.         printf("从对首到队尾所有的排队病人的病历号为: \n");
30.         Q.outputQueue();
31.         cout<<endl;
32.         break;
33.     }
34.     case 4:{
35.         printf("从对首到队尾所有的排队病人的病历号为: \n");
36.         Q.outputQueue();
37.         cout<<endl;
38.         for(int i=0; i<=Q.getSize(); ++i)
39.             Q.dequeue();
40.         if(Q.isEmpty())
41.             cout<<"已全部就诊"<<endl;
42.         break;
43.     }
44.     case 5:{
45.         Q.~Queue();
46.         exit(0);
47.     }
48.     default:
49.         break;
50.     }
51. }
52. }
53.
54. int main()
55. {
56.     Queue Q;
57.     menu(Q);
58. }

```

运行截图

```

F:\QT\setup\Tools\QtCreator\bin\qtcreator_process_stub.exe
菜单:
1. 排队
2. 就诊
3. 查看排队
4. 不再排队, 余下依次就诊
5. 下班
请输入要选择的操作: 1
请输入病历号: 1
1号病人入队成功
请输入要选择的操作: 1
请输入病历号: 2
2号病人入队成功
请输入要选择的操作: 1
请输入病历号: 3
3号病人入队成功
请输入要选择的操作: 3
从对首到队尾所有的排队病人的病历号为:
1 2 3
请输入要选择的操作: 2
1号病人出队成功
请输入要选择的操作: 3
从对首到队尾所有的排队病人的病历号为:
2 3
请输入要选择的操作: 4
从对首到队尾所有的排队病人的病历号为:
2 3
2号病人出队成功
3号病人出队成功
已全部就诊
请输入要选择的操作: 5
    
```

实验三 串和数组实验

【实验目的】

1. 掌握串的顺序存储结构
2. 掌握串的基本算法及应用
3. 掌握模式匹配的各种算法
4. 掌握数组和广义表的基本算法

【实验学时】

8 学时

【实验内容】

实验内容题目二：KMP

1. 编写一个程序，利用 KMP 算法求子串 t 在主串 s 中出现的次数，并以 s="aaabbdabbde", t="aabb" 为例，显示其匹配过程。（匹配过程的显示选做）。

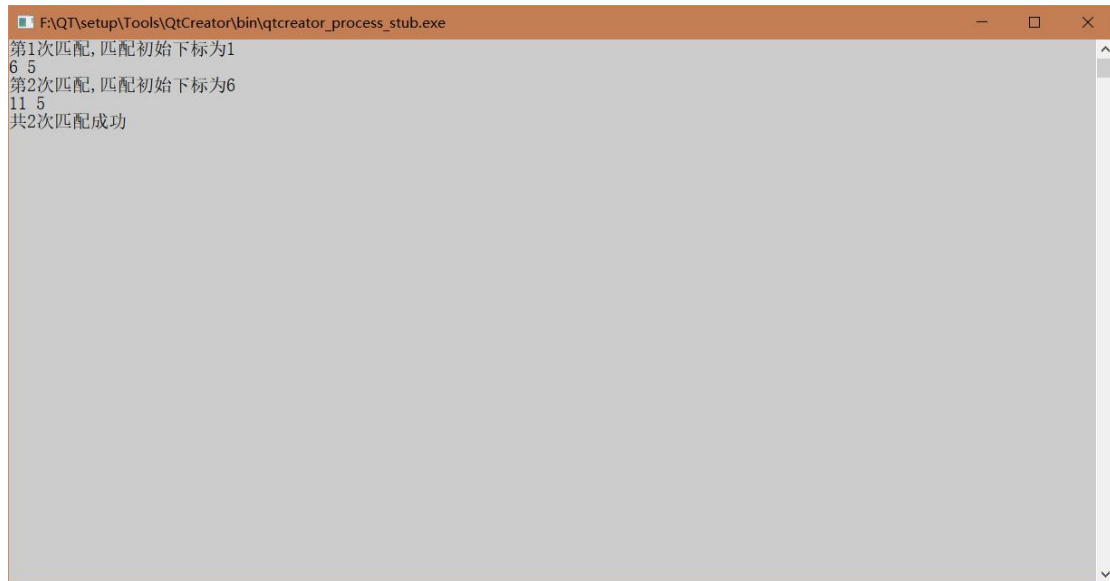
代码

```
1. #include <iostream>
2. //using namespace std;
3. int next[100];
4.
5. //返回匹配次数
6. int KMP(std::string s, std::string t){
7.     int ls = s.size();
8.     int lt = t.size();
9.     int i=0,j=0;
10.    int flag = 0;    //记录匹配成功次数
11.    while(i<ls){
12.        //失配则回溯 相等则继续
13.        if(j== -1 || s[i]==t[j]){
14.            i++;
15.            j++;
16.        }else
17.            j = next[j];
18.        //如果成功匹配,继续进行下一次匹配
19.        if(j == lt){
20.            flag++;
21.            printf("第%d 次匹配,匹配初始下标为%d\n",flag,i-lt);
22.            std::cout<<i<<" "<<j<<std::endl;
23.            i = i-lt+1;
```



```
24.         j=0;
25.     }
26. }
27.     return flag;
28. }
29.
30. void getNext(std::string t){
31.     int l = t.size();
32.     next[0] = -1; //初始化
33.     int i=0, j=-1;
34.     for(i=0; i<l; ){
35.         //一直回溯 j 直到 t[i]==t[j]或 j 减小到-1
36.         if(j==-1 || t[i]==t[j]){
37.             i++;
38.             j++;
39.             next[i] = j;
40.         }else
41.             j = next[j];
42.     }
43. }
44.
45. int main(){
46.     std::string s = "aaabbdabbde";
47.     std::string t = "aabb";
48.     getNext(t);
49.     int time = KMP(s,t);
50.     printf("共%d 次匹配成功",time);
51. }
```

运行截图



实验内容题目三：稀疏矩阵

2. 实现稀疏矩阵的基本运算。假设 $n \times n$ 的稀疏矩阵 A 采用三元组表示，设计一个程序，实现如下功能：

- (1) 生成如下两个稀疏矩阵的三元组 a 和 b；

$$\begin{pmatrix} 1 & 0 & 3 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix} \quad \begin{pmatrix} 3 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 2 \end{pmatrix}$$

- (2) 输出 a 转置矩阵的三元组
 (3) 输出 a+b 的三元组

代码

sparsematrix.h

```

1. #ifndef SPARSEMATRIX_H
2. #define SPARSEMATRIX_H
3.
4. #include <iostream>
5.
6. //三元组类 <row, col, value>
7. struct triple
8. {
9.     int row, col;    //行号, 列号
10.    int value; //元素
11.    //构造
    
```

```

12.     triple(){}
13.     triple(int row, int col, int value){
14.         this->col = col;
15.         this->row = row;
16.         this->value = value;
17.     }
18.     triple &operator = (triple & tri);
19. };
20.
21. //三元组表
22. class sparseMatrix
23. {
24. private:
25.     int rows, cols; //行数, 列数
26.     int size;      //项数
27.     triple *data = new triple[16]; //每一个三元组
28.     int maxSize=16; //最大存储容量
29.
30. public:
31.     sparseMatrix(); //构造
32.     sparseMatrix(sparseMatrix &spa); //构造
33.     ~sparseMatrix(){} //析构
34.     void setData(int row, int col, int value){data[size].col=col; data[size].
        row=row; data[size].value=value; size++;} //设置项方式 2
35.     sparseMatrix transpose(); //转置
36.     sparseMatrix add(sparseMatrix &spa); //加法
37.     void output(); //输出
38.     sparseMatrix & operator = (sparseMatrix & spa); //重写=
39.
40. };
41.
42. #endif // SPARSEMATRIX_H

```

sparsematrix.cpp

```

1. #include "sparsematrix.h"
2.
3. triple& triple::operator=(triple &tri){
4.     this->col = tri.col;
5.     this->row = tri.row;
6.     this->value = tri.value;
7.     return *this;
8. }
9.

```

```

10. sparseMatrix::sparseMatrix()
11. {
12.     maxSize = 16;    //4*4
13.     size = 0;
14.     cols = 4;
15.     rows = 4;
16.     data = new triple[maxSize];
17. }
18.
19. sparseMatrix::sparseMatrix(sparseMatrix &spa){
20.     this->rows = spa.rows;
21.     this->cols = spa.cols;
22.     this->maxSize = spa.maxSize;
23.     data = new triple[maxSize];
24.     for(int i=0; i<size; ++i){
25.         data[i].col = spa.data[i].col;
26.         data[i].row = spa.data[i].row;
27.         data[i].value = spa.data[i].value;
28.     }
29. }
30.
31. sparseMatrix sparseMatrix::transpose(){
32.     sparseMatrix tra;    //对该表进行操作
33.     tra.cols = this->cols;
34.     tra.rows = this->rows;
35.     tra.size = this->size;
36.     tra.maxSize = this->maxSize;
37.     if(size>0){
38.         int a = 0;
39.         for(int i=0; i<cols; ++i)    //按列扫描
40.             for(int j=0; j<size; ++j)    //找第 i 列三元组
41.                 if(data[j].col == i){
42.                     //逐列转置
43.                     tra.data[a].row = i;
44.                     tra.data[a].col = data[j].row;
45.                     tra.data[a].value = data[j].value;
46.                     a++;
47.                 }
48.     }
49.     return tra;
50. }
51.
52. sparseMatrix sparseMatrix::add(sparseMatrix &spa){
53.     sparseMatrix s;

```

```

54.     if(this->cols!=spa.cols||this->rows!=spa.rows){
55.         printf("矩阵不匹配");
56.         exit(0);
57.     }
58.     s.cols = this->cols;
59.     s.rows = this->rows;
60.     s.maxSize = this->maxSize;
61.     s.size = 0;
62.     int index_this, index_spa;
63.     int i,j;
64.     for(i=0,j=0; i<size&&j<spa.size; ){
65.         index_this = cols*(data[i].row) + data[i].col;
66.         index_spa = cols*(spa.data[j].row) + spa.data[j].col;
67.
68.         //data[i]在 spa.data[j]前--插入
69.         if(index_this < index_spa){
70.             s.data[s.size] = data[i];
71.             i++;
72.         }
73.         //data[i]在 spa.data[j]后--插入
74.         else if(index_this > index_spa){
75.             s.data[s.size] = spa.data[j];
76.             j++;
77.         }
78.         //data[i]与 spa.data[j]位置相同--相加
79.         else{
80.             s.data[s.size] = data[i];
81.             s.data[s.size].value = data[i].value + spa.data[j].value;
82.             i++;
83.             j++;
84.         }
85.         s.size++;
86.     }
87.     //复制剩余元素
88.     for( ; i<this->size; ++i){
89.         s.data[s.size] = data[i];
90.         s.size++;
91.     }
92.     for( ; j<spa.size; ++j){
93.         s.data[s.size] = spa.data[j];
94.         s.size++;
95.     }
96.     return s;
97. }

```

```

98.
99. void sparseMatrix::output(){
100.     for(int i=0; i<cols; ++i){
101.         for(int j=0; j<rows; ++j){
102.             int flag = 0;    //记录是否为已有项
103.             for(int k=0; k<size; ++k){
104.                 if(data[k].row==j && data[k].col==i){
105.                     printf("%d ",data[k].value);
106.                     flag = 1;
107.                 }
108.             }
109.             if(flag == 0)
110.                 printf("0 ");
111.         }
112.         printf("\n");
113.     }
114. }

```

main.cpp

```

1. #include <iostream>
2. #include "sparsematrix.h"
3.
4. int main()
5. {
6.     //生成两个稀疏矩阵 a 和 b
7.     sparseMatrix a;
8.     sparseMatrix b;
9.
10.    //该顺序不能变
11.    //按 4*(row) + col 大小排序
12.    a.setData(0,0,1);
13.    a.setData(1,1,1);
14.    a.setData(0,2,3);
15.    a.setData(2,2,1);
16.    a.setData(3,2,1);
17.    a.setData(3,3,1);
18.
19.    b.setData(0,0,3);
20.    b.setData(1,1,4);
21.    b.setData(2,2,1);
22.    b.setData(3,3,2);
23.
24.    printf("矩阵 a: \n");

```

```

25.     a.output();
26.     printf("矩阵 b: \n");
27.     b.output();
28.
29.     //输出 a 转置的三元组
30.     printf("a 转置的三元组: \n");
31.     a.transpose().output();
32.
33.     //输出 a+b 的三元组
34.     printf("a+b 的三元组: \n");
35.     a.add(b).output();
36. }

```

运行截图

```

F:\QT\setup\Tools\QtCreator\bin\qtcreator_process_stub.exe
矩阵a:
1 0 0 0
0 1 0 0
3 0 1 1
0 0 0 1
矩阵b:
3 0 0 0
0 4 0 0
0 0 1 0
0 0 0 2
a转置的三元组:
1 0 3 0
0 1 0 0
0 0 1 0
0 0 1 1
a+b的三元组:
4 0 0 0
0 5 0 0
3 0 2 1
0 0 0 3

```

实验内容题目五：马鞍点

- 求一个矩阵中的马鞍点。如果矩阵 A 中存在这样的元素，满足条件： $A[i][j]$ 是第 i 行中值最小的元素，并且是第 j 列中值最大的元素，则称之为该矩阵的一个马鞍点。设计一个程序，求矩阵中所有的马鞍点。

代码

```

1. #include <iostream>
2. using namespace std;
3.
4. int main(){
5.     int data[100][100];

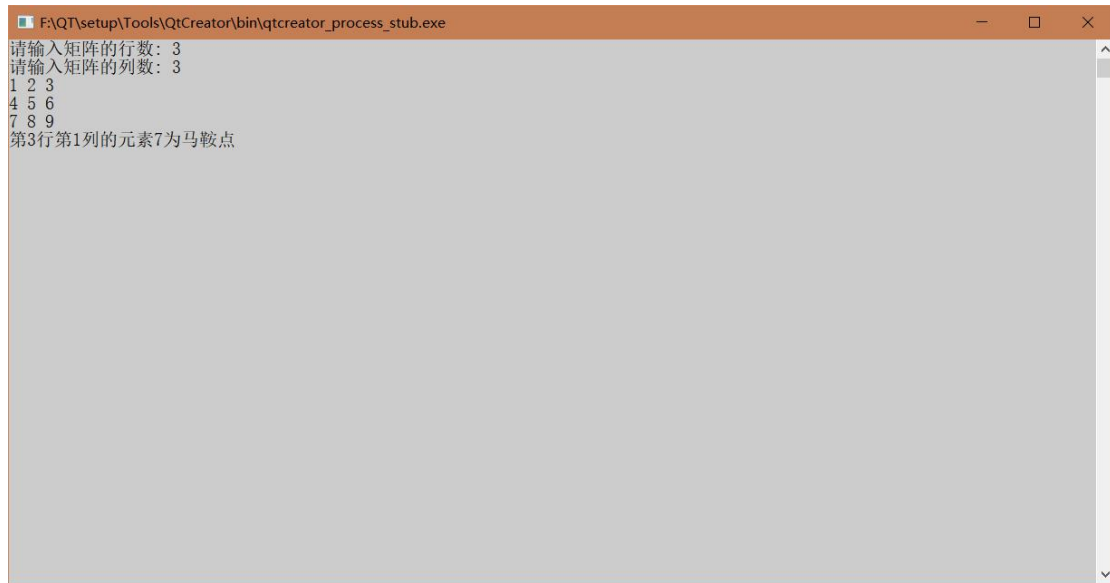
```

```

6.     int rows,cols;
7.     printf("请输入矩阵的行数: ");
8.     scanf("%d",&rows);
9.     printf("请输入矩阵的列数: ");
10.    scanf("%d",&cols);
11.
12.    for(int i=0; i<rows; ++i)
13.        for(int j=0; j<cols; ++j)
14.            cin>>data[i][j];
15.
16.    for(int i=0; i<rows; ++i){
17.        int minRow = data[i][0];
18.        int minRowJ = 0;
19.        for(int j=0; j<cols; ++j)
20.            if(data[i][j]<minRow){
21.                minRow = data[i][j];
22.                minRowJ = j;
23.            }
24.        int flag = 1;
25.        for(int k=0; k<rows; ++k){
26.            if(data[k][minRowJ]>minRow){
27.                flag = 0;
28.                break;
29.            }
30.        }
31.        if(flag)
32.            printf("第%d 行第%d 列的元素%d 为马鞍点",i+1,minRowJ+1,minRow);
33.        else
34.            continue;
35.    }
36. }

```

运行截图



A screenshot of a Qt Creator console window. The title bar shows the file path 'F:\QT\setup\Tools\QtCreator\bin\qtcreator_process_stub.exe'. The console output is as follows:

```
请输入矩阵的行数: 3  
请输入矩阵的列数: 3  
1 2 3  
4 5 6  
7 8 9  
第3行第1列的元素7为马鞍点
```

实验四 树实验

【实验目的】

1. 掌握二叉树的顺序和链式存储结构
2. 掌握二叉树的建立、遍历、线索化等基本算法及应用
3. 掌握哈夫曼树的构造过程和哈夫曼编码方法
4. 了解树、森林与二叉树的转换算法
5. 了解哈夫曼编码和解码算法

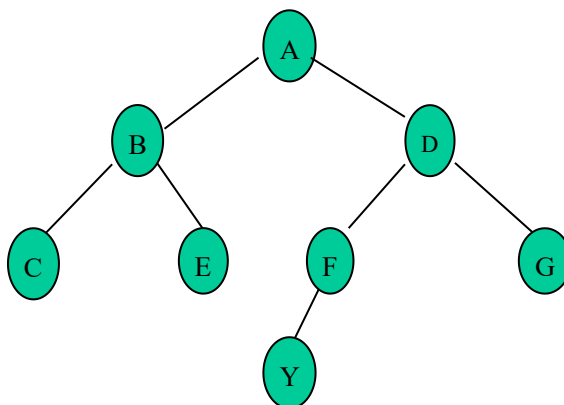
【实验学时】

10 学时

【实验内容】

实验内容题目一：二叉树

1. 编写一个程序，实现二叉树的各种运算，并在此基础上设计一个主程序完成如下功能：



图一

- (1) 由如图一所示的二叉树创建其对应的二叉链表存储结构（提示：由二叉树的扩展前序序列构造二叉链表）
- (2) 输出二叉树的**中序**、先序、后序遍历序列的递归和非递归算法（中序必做，先序和后序选做）；
- (3) 输出二叉树 B（树形结构或者广义表形式）；（选做）
- (4) 输出二叉树中指定结点值（假设所有节点值不同）的左右孩子结点；
- (5) 输出二叉树 B 的叶子结点个数；
- (6) 输出该二叉树的高度；
- (7) 输出二叉树中指定结点值（假设所有节点值不同）的结点所在的层次；（选做）
- (8) 释放二叉树 B。

代码

binarytree.h

```

1. #ifndef BINARYTREE_H
2. #define BINARYTREE_H
3.
4. #include <iostream>
5.
6. struct binaryTreeNode{
7.     char data; //元素
8.     binaryTreeNode *leftChild, *rightChild; //左孩子,右孩子
9.     binaryTreeNode(){leftChild = NULL; rightChild = NULL;}
10.    binaryTreeNode(char d){
11.        data = d;
12.        leftChild = NULL;
13.        rightChild = NULL;
14.    }
15. };
16.
17. class binaryTree
18. {
19. protected:
20.
21.     char end; //输入终止符
22.     binaryTreeNode *parent(binaryTreeNode *subTree, binaryTreeNode*d); //返回父结点
23.     int size(binaryTreeNode *subTree); //返回结点个数
24.     int height(binaryTreeNode *subTree); //返回高度
25.     void Insert(binaryTreeNode *subTree, char d); //插入
26.     void destory(binaryTreeNode *subTree); //删除
27.     bool find(binaryTreeNode *subTree, char d); //查找
28.
29. public:
30.     binaryTreeNode *root; //根
31.     binaryTree(){root = NULL;}
32.     binaryTree(char end){this->end = end; root = NULL;}
33.     ~binaryTree(){destory(root);}
34.
35.     void createTreeByPreOrder(binaryTreeNode * &subTree); //前序遍历创建树
36.
37.     binaryTreeNode *parent(binaryTreeNode *d){
38.         if(root!=NULL && root!=d)
39.             return parent(root, d);
40.         else
41.             return NULL;

```

```

42.     } //返回父结点
43.     binaryTreeNode *leftChild(binaryTreeNode *d){
44.         if(d!=NULL)
45.             return d->leftChild;
46.         else
47.             return NULL;
48.     } //返回左孩子
49.     binaryTreeNode *rightChild(binaryTreeNode *d){
50.         if(d!=NULL)
51.             return d->rightChild;
52.         else
53.             return NULL;
54.     } //返回右孩子
55.
56.     bool isEmpty(){return root==NULL;} //判断是否为空
57.     binaryTreeNode *getRoot(){return root;} //返回根结点
58.     char getData(binaryTreeNode *tree){return tree->data;}
59.     int getSize(){return size(root);} //获取结点数
60.     int getLeaves(); //获取叶子结点
61.     int getHeight(){return height(root);} //获取高度
62.     binaryTreeNode *getNodeByData(char d);
63.
64.     //递归算法
65.     void preOrder_recursion(binaryTreeNode *subTree); //前序遍历
66.     void infixOrder_recursion(binaryTreeNode *subTree); //中序遍历
67.     void postOrder_recursion(binaryTreeNode *subTree); //后序遍历
68.
69.     //非递归算法
70.     void preOrder_NONrecursion(binaryTreeNode *subTree); //前序遍历
71.     void infixOrder_NONrecursion(binaryTreeNode *subTree); //中序遍历
72.
73.     void output(binaryTreeNode *b);
74.
75. };
76.
77. #endif // BINARYTREE_H

```

binarytree.cpp

```

1. #include "binarytree.h"
2. #include "stack"
3.
4. void binaryTree::createTreeByPreOrder(binaryTreeNode * &subTree){
5.     char c;

```

```

6.     std::cin>>c;
7.     if(c!=end){
8.         //printf("%c",end);
9.         subTree = new binaryTreeNode(c);
10.        //subTree->data = c;
11.        createTreeByPreOrder(subTree->leftChild);
12.        createTreeByPreOrder(subTree->rightChild);
13.    }else{
14.        subTree = NULL;
15.    }
16. }
17.
18. void binaryTree::destory(binaryTreeNode *subTree){
19.     if(subTree != NULL){
20.         destory(subTree->leftChild);
21.         destory(subTree->rightChild);
22.         delete subTree;
23.     }
24. }
25.
26. binaryTreeNode * binaryTree::parent(binaryTreeNode *subTree, binaryTreeNode
    *d){
27.     //不存在则返回空
28.     if(subTree == NULL)
29.         return NULL;
30.
31.     //找到则返回
32.     if(subTree->leftChild == d || subTree->rightChild == d)
33.         return subTree;
34.
35.     //递归搜索
36.     binaryTreeNode *search = parent(subTree->leftChild);
37.     if(search != NULL)
38.         return search;
39.     else
40.         return parent(subTree->rightChild,d);
41. }
42.
43. //递归算法
44. void binaryTree::preOrder_recursion(binaryTreeNode *subTree){
45.     if(subTree != NULL){
46.         printf("%c",subTree->data);
47.         preOrder_recursion(subTree->leftChild);
48.         preOrder_recursion(subTree->rightChild);

```

```

49.     }
50. }
51.
52. void binaryTree::infixOrder_recursion(binaryTreeNode *subTree){
53.     if(subTree != NULL){
54.         infixOrder_recursion(subTree->leftChild);
55.         printf("%c",subTree->data);
56.         infixOrder_recursion(subTree->rightChild);
57.     }
58. }
59.
60. void binaryTree::postOrder_recursion(binaryTreeNode *subTree){
61.     if(subTree != NULL){
62.         postOrder_recursion(subTree->leftChild);
63.         postOrder_recursion(subTree->rightChild);
64.         printf("%c",subTree->data);
65.     }
66. }
67.
68. //非递归算法
69. void binaryTree::preOrder_NONrecursion(binaryTreeNode *subTree){
70.     std::stack<binaryTreeNode *> s;
71.     binaryTreeNode *bin = root;
72.     s.push(NULL);
73.     while(bin!=NULL){
74.         printf("%c",bin->data);
75.         if(bin->rightChild != NULL)
76.             s.push(bin->rightChild);
77.         if(bin->leftChild != NULL)
78.             bin = bin->leftChild;
79.         else{
80.             bin = s.top();
81.             s.pop();
82.         }
83.     }
84. }
85.
86. void binaryTree::infixOrder_NONrecursion(binaryTreeNode *subTree){
87.     std::stack<binaryTreeNode *> s;
88.     binaryTreeNode * bin = root;
89.     do{
90.         while(bin != NULL){
91.             s.push(bin);
92.             bin = bin->leftChild;

```

```

93.     }
94.     if(!s.empty()){
95.         bin = s.top();
96.         s.pop();
97.         printf("%c",bin->data);
98.         bin = bin->rightChild;
99.     }
100. }while(bin!=NULL || !s.empty());
101. }
102.
103. binaryTreeNode *binaryTree::getNodeByData(char d){
104.     std::stack<binaryTreeNode *> s;
105.     binaryTreeNode *bin = root;
106.     int flag = 0;
107.     s.push(NULL);
108.     while(bin!=NULL){
109.         if(bin->data == d){
110.             flag = 1;
111.             break;
112.         }
113.         if(bin->rightChild != NULL)
114.             s.push(bin->rightChild);
115.         if(bin->leftChild != NULL)
116.             bin = bin->leftChild;
117.         else{
118.             bin = s.top();
119.             s.pop();
120.         }
121.     }
122.     if(flag)
123.         return bin;
124.     else
125.         return NULL;
126. }
127.
128. int binaryTree::size(binaryTreeNode *subTree){
129.     if(subTree == NULL)
130.         return 0;
131.     else
132.         return size(subTree->leftChild)+size(subTree->rightChild)+1;
133. }
134.
135. int binaryTree::getLeaves(){
136.     int size = 0;

```

```

137.     std::stack<binaryTreeNode *> s;
138.     binaryTreeNode * bin = root;
139.     do{
140.         while(bin != NULL){
141.             s.push(bin);
142.             bin = bin->leftChild;
143.         }
144.         if(!s.empty()){
145.             bin = s.top();
146.             s.pop();
147.             if(bin->rightChild == NULL && bin->leftChild == NULL)
148.                 size++;
149.             bin = bin->rightChild;
150.         }
151.     }while(bin!=NULL || !s.empty());
152.     return size;
153.
154. }
155.
156. int binaryTree::height(binaryTreeNode *subTree){
157.     if(subTree == NULL)
158.         return 0;
159.     else{
160.         int i = height(subTree->leftChild);
161.         int j = height(subTree->rightChild);
162.         return i<j?j+1:i+1;
163.     }
164. }
165.
166.
167.
168. void binaryTree::output(binaryTreeNode *b){
169.     if(b != NULL){
170.         std::cout<<b->data;
171.         if(b->leftChild!=NULL||b->rightChild!=NULL){
172.             std::cout<<'(';
173.             output(b->leftChild);
174.             std::cout<<",";
175.             if(b->rightChild!=NULL)
176.                 output(b->rightChild);
177.             std::cout<<")";
178.         }
179.     }
180. }

```


main.cpp

```

1. #include <iostream>
2. #include "binarytree.h"
3. using namespace std;
4.
5. int main()
6. {
7.     //拓展先序序列创建树
8.     char end = '*';
9.     binaryTree b(end);
10.    printf("请输入所要创建的树的前序遍历:\n");
11.    b.createTreeByPreOrder(b.root);
12.    cout<<"创建完成"<<endl<<endl;
13.
14.    //输出
15.    //递归
16.    //前序
17.    printf("递归先序输出: ");
18.    b.preOrder_recursion(b.getRoot());
19.    cout<<endl;
20.    //中序
21.    printf("递归中序输出: ");
22.    b.infixOrder_recursion(b.getRoot());
23.    cout<<endl;
24.    //后续
25.    printf("递归后序输出: ");
26.    b.postOrder_recursion(b.getRoot());
27.    cout<<endl;
28.    //非递归
29.    //前序
30.    printf("非递归先序输出: ");
31.    b.preOrder_NONrecursion(b.getRoot());
32.    cout<<endl;
33.    //中序
34.    printf("非递归中序输出: ");
35.    b.infixOrder_NONrecursion(b.getRoot());
36.    cout<<endl<<endl;
37.
38.    //广义表输出
39.    cout<<"广义表输出: ";
40.    b.output(b.getRoot());
41.    cout<<endl<<endl;
42.

```

```

43. //输出二叉树中指定结点值的左右孩子结点
44. char data;
45. cout<<"请输入要查找左右孩子的结点的结点值: ";
46. cin>>data;
47. binaryTreeNode *tmp = b.getNodeByData(data);
48. if(b.leftChild(tmp)!=NULL)
49.     cout<<"左孩子:"<<b.leftChild(tmp)->data<<endl;
50. else
51.     cout<<"无左孩子"<<endl;
52. if(b.rightChild(tmp)!=NULL)
53.     cout<<"右孩子:"<<b.rightChild(tmp)->data<<endl;
54. else
55.     cout<<"无右孩子"<<endl;
56. cout<<endl;
57.
58. //输出二叉树叶子结点个数
59. cout<<"二叉树叶子结点个数: "<<b.getLeaves()<<endl<<endl;
60.
61. //输出二叉树高度
62. cout<<"二叉树高度: "<<b.getHeight();
63.
64. //释放二叉树
65. b.~binaryTree();
66. }

```

运行截图

```

F:\QT\setup\Tools\QtCreator\bin\qtcreator_process_stub.exe
请输入所要创建的树的前序遍历:
ABC**E**DFY***G**
创建完成

递归先序输出: ABCEDFYG
递归中序输出: CBEAYFDG
递归后序输出: CEBYFGDA
非递归先序输出: ABCEDFYG
非递归中序输出: CBEAYFDG

广义表输出: A(B(C, E), D(F(Y, ), G))

请输入要查找左右孩子的结点的结点值: C
无左孩子
无右孩子

二叉树叶子结点个数: 4
二叉树高度: 4

```

实验内容题目二：前序中序构造二叉树

2. 已知一棵树的前序遍历序列和中序遍历序列，试构造这棵二叉树，以 P₂₄₈ 的 5.18 验证。

代码

binarytree.h

```

1. #ifndef BINARYTREE_H
2. #define BINARYTREE_H
3.
4. #include <iostream>
5.
6. struct binaryTreeNode{
7.     char data; //元素
8.     binaryTreeNode *leftChild, *rightChild; //左孩子,右孩子
9.     binaryTreeNode(){leftChild = NULL; rightChild = NULL;}
10.    binaryTreeNode(char d){
11.        data = d;
12.        leftChild = NULL;
13.        rightChild = NULL;
14.    }
15. };
16.
17. class binaryTree
18. {
19. protected:
20.
21.     char end; //输入终止符
22.     binaryTreeNode *parent(binaryTreeNode *subTree, binaryTreeNode*d); //返回父结点
23.     int size(binaryTreeNode *subTree); //返回结点个数
24.     int height(binaryTreeNode *subTree); //返回高度
25.     void Insert(binaryTreeNode *subTree, char d); //插入
26.     void destory(binaryTreeNode *subTree); //删除
27.     bool find(binaryTreeNode *subTree, char d); //查找
28.
29. public:
30.     binaryTreeNode *root; //根
31.     binaryTree(){root = NULL;}
32.     binaryTree(char end){this->end = end; root = NULL;}
33.     ~binaryTree(){destory(root);}
34.
35.     void createTreeByPreOrder(binaryTreeNode * &subTree); //前序遍历创建树
36.     binaryTreeNode * createTreeByPreOrderAndInfixOrder

```

```

37.     (int preFront, int preRear, int infFront, int infRear, char preOrder[],
      char infixOrder[]); //根据先序中序创建树
38.
39.     binaryTreeNode *parent(binaryTreeNode *d){
40.         if(root!=NULL && root!=d)
41.             return parent(root, d);
42.         else
43.             return NULL;
44.     } //返回父结点
45.     binaryTreeNode *leftChild(binaryTreeNode *d){
46.         if(d!=NULL)
47.             return d->leftChild;
48.         else
49.             return NULL;
50.     } //返回左孩子
51.     binaryTreeNode *rightChild(binaryTreeNode *d){
52.         if(d!=NULL)
53.             return d->rightChild;
54.         else
55.             return NULL;
56.     } //返回右孩子
57.
58.     bool isEmpty(){return root==NULL;} //判断是否为空
59.     binaryTreeNode *getRoot(){return root;} //返回根结点
60.     char getData(binaryTreeNode *tree){return tree->data;}
61.     int getSize(){return size(root);} //获取结点数
62.     int getLeaves(); //获取叶子结点
63.     int getHeight(){return height(root);} //获取高度
64.     binaryTreeNode *getNodeByData(char d);
65.
66.     //递归算法
67.     void preOrder_recursion(binaryTreeNode *subTree); //前序遍历
68.     void infixOrder_recursion(binaryTreeNode *subTree); //中序遍历
69.     void postOrder_recursion(binaryTreeNode *subTree); //后序遍历
70.
71.     //非递归算法
72.     void preOrder_NONrecursion(binaryTreeNode *subTree); //前序遍历
73.     void infixOrder_NONrecursion(binaryTreeNode *subTree); //中序遍历
74.
75.     void output(binaryTreeNode *b);
76.
77. };
78.
79. #endif // BINARYTREE_H

```

binarytree.cpp

```

1. #include "binarytree.h"
2. #include "stack"
3.
4. void binaryTree::createTreeByPreOrder(binaryTreeNode * &subTree){
5.     char c;
6.     std::cin>>c;
7.     if(c!=end){
8.         //printf("%c",end);
9.         subTree = new binaryTreeNode(c);
10.        //subTree->data = c;
11.        createTreeByPreOrder(subTree->leftChild);
12.        createTreeByPreOrder(subTree->rightChild);
13.    }else{
14.        subTree = NULL;
15.    }
16. }
17.
18. binaryTreeNode * binaryTree::createTreeByPreOrderAndInfixOrder
19. (int preFront, int preRear, int infFront, int infRear, char preOrder[], char
    infixOrder[]){
20.     if(preFront>preRear||infFront>infRear)
21.         return NULL;
22.     binaryTreeNode *subTree = new binaryTreeNode;
23.     subTree->data = preOrder[preFront];    //根节点
24.     int flag=0;
25.     for(int i=infFront; i<=infRear; ++i){
26.         if(preOrder[preFront] == infixOrder[i]){
27.             flag = i;
28.             break;
29.         }
30.     }
31.     subTree->leftChild = createTreeByPreOrderAndInfixOrder(preFront+1,preFro
    nt+flag-infFront,infFront,flag-1,preOrder,infixOrder);
32.     subTree->rightChild = createTreeByPreOrderAndInfixOrder(preFront+flag-in
    fFront+1,preRear,flag+1,infRear,preOrder,infixOrder);
33.
34.     return subTree;
35. }
36.
37. void binaryTree::destory(binaryTreeNode *subTree){
38.     if(subTree != NULL){
39.         destory(subTree->leftChild);

```

```

40.         destory(subTree->rightChild);
41.         delete subTree;
42.     }
43. }
44.
45. binaryTreeNode * binaryTree::parent(binaryTreeNode *subTree, binaryTreeNode
    *d){
46.     //不存在则返回空
47.     if(subTree == NULL)
48.         return NULL;
49.
50.     //找到则返回
51.     if(subTree->leftChild == d || subTree->rightChild == d)
52.         return subTree;
53.
54.     //递归搜索
55.     binaryTreeNode *search = parent(subTree->leftChild);
56.     if(search != NULL)
57.         return search;
58.     else
59.         return parent(subTree->rightChild,d);
60. }
61.
62. //递归算法
63. void binaryTree::preOrder_recursion(binaryTreeNode *subTree){
64.     if(subTree != NULL){
65.         printf("%c",subTree->data);
66.         preOrder_recursion(subTree->leftChild);
67.         preOrder_recursion(subTree->rightChild);
68.     }
69. }
70.
71. void binaryTree::infixOrder_recursion(binaryTreeNode *subTree){
72.     if(subTree != NULL){
73.         infixOrder_recursion(subTree->leftChild);
74.         printf("%c",subTree->data);
75.         infixOrder_recursion(subTree->rightChild);
76.     }
77. }
78.
79. void binaryTree::postOrder_recursion(binaryTreeNode *subTree){
80.     if(subTree != NULL){
81.         postOrder_recursion(subTree->leftChild);
82.         postOrder_recursion(subTree->rightChild);

```

```

83.         printf("%c",subTree->data);
84.     }
85. }
86.
87. //非递归算法
88. void binaryTree::preOrder_NONrecursion(binaryTreeNode *subTree){
89.     std::stack<binaryTreeNode *> s;
90.     binaryTreeNode *bin = root;
91.     s.push(NULL);
92.     while(bin!=NULL){
93.         printf("%c",bin->data);
94.         if(bin->rightChild != NULL)
95.             s.push(bin->rightChild);
96.         if(bin->leftChild != NULL)
97.             bin = bin->leftChild;
98.         else{
99.             bin = s.top();
100.            s.pop();
101.        }
102.    }
103. }
104.
105. void binaryTree::infixOrder_NONrecursion(binaryTreeNode *subTree){
106.     std::stack<binaryTreeNode *> s;
107.     binaryTreeNode * bin = root;
108.     do{
109.         while(bin != NULL){
110.             s.push(bin);
111.             bin = bin->leftChild;
112.         }
113.         if(!s.empty()){
114.             bin = s.top();
115.             s.pop();
116.             printf("%c",bin->data);
117.             bin = bin->rightChild;
118.         }
119.     }while(bin!=NULL || !s.empty());
120. }
121.
122. binaryTreeNode *binaryTree::getNodeByData(char d){
123.     std::stack<binaryTreeNode *> s;
124.     binaryTreeNode *bin = root;
125.     int flag = 0;
126.     s.push(NULL);

```

```

127.     while(bin!=NULL){
128.         if(bin->data == d){
129.             flag = 1;
130.             break;
131.         }
132.         if(bin->rightChild != NULL)
133.             s.push(bin->rightChild);
134.         if(bin->leftChild != NULL)
135.             bin = bin->leftChild;
136.         else{
137.             bin = s.top();
138.             s.pop();
139.         }
140.     }
141.     if(flag)
142.         return bin;
143.     else
144.         return NULL;
145. }
146.
147. int binaryTree::size(binaryTreeNode *subTree){
148.     if(subTree == NULL)
149.         return 0;
150.     else
151.         return size(subTree->leftChild)+size(subTree->rightChild)+1;
152. }
153.
154. int binaryTree::height(binaryTreeNode *subTree){
155.     if(subTree == NULL)
156.         return 0;
157.     else{
158.         int i = height(subTree->leftChild);
159.         int j = height(subTree->rightChild);
160.         return i<j?j+1:i+1;
161.     }
162. }
163.
164. int binaryTree::getLeaves(){
165.     int size = 0;
166.     std::stack<binaryTreeNode *> s;
167.     binaryTreeNode * bin = root;
168.     do{
169.         while(bin != NULL){
170.             s.push(bin);

```



```

171.         bin = bin->leftChild;
172.     }
173.     if(!s.empty()){
174.         bin = s.top();
175.         s.pop();
176.         if(bin->rightChild == NULL && bin->leftChild == NULL)
177.             size++;
178.         bin = bin->rightChild;
179.     }
180. }while(bin!=NULL || !s.empty());
181. return size;
182.
183. }
184.
185. void binaryTree::output(binaryTreeNode *b){
186.     if(b != NULL){
187.         std::cout<<b->data;
188.         if(b->leftChild!=NULL||b->rightChild!=NULL){
189.             std::cout<<'(';
190.             output(b->leftChild);
191.             std::cout<<",";
192.             if(b->rightChild!=NULL)
193.                 output(b->rightChild);
194.             std::cout<<")";
195.         }
196.     }
197. }

```

main.cpp

```

1. #include <iostream>
2. #include "binarytree.h"
3. using namespace std;
4.
5. int main()
6. {
7.     //拓展先序序列创建树
8.     char end = '*';
9.     binaryTree b(end);
10.    char preOrder[100]={0};
11.    char infOrder[100]={0};
12.    int length;
13.    printf("请出入序列长度: ");
14.    cin>>length;

```

```

15.     printf("请输入所要创建的树的前序遍历:\n");
16.     for(int i=0; i<length; ++i)
17.         cin>>preOrder[i];
18.     printf("请输入所要创建的树的中序遍历:\n");
19.     for(int i=0; i<length; ++i)
20.         cin>>infOrder[i];
21.
22. //     int length = 8;
23. //     char preOrder[]={'A','B','C','E','D','F','Y','G'};
24. //     char infOrder[]={'C','B','E','A','Y','F','D','G'};
25.
26.     b.root = b.createTreeByPreOrderAndInfixOrder(0,length-1,0,length-1,preOr
        der,infOrder);
27.     cout<<"创建完成"<<endl<<endl;
28.
29.     //输出
30.     //递归
31.     //前序
32.     printf("递归先序输出: ");
33.     b.preOrder_recursion(b.getRoot());
34.     cout<<endl;
35.     //中序
36.     printf("递归中序输出: ");
37.     b.infixOrder_recursion(b.getRoot());
38.     cout<<endl;
39.     //后续
40.     printf("递归后序输出: ");
41.     b.postOrder_recursion(b.getRoot());
42.     cout<<endl;
43.     //非递归
44.     //前序
45.     printf("非递归先序输出: ");
46.     b.preOrder_NONrecursion(b.getRoot());
47.     cout<<endl;
48.     //中序
49.     printf("非递归中序输出: ");
50.     b.infixOrder_NONrecursion(b.getRoot());
51.     cout<<endl<<endl;
52.
53.     //输出二叉树中指定结点值的左右孩子结点
54.     char data;
55.     cout<<"请输入要查找左右孩子的结点的结点值: ";
56.     cin>>data;
57.     binaryTreeNode *tmp = b.getNodeByData(data);

```

```

58.     if(b.leftChild(tmp)!=NULL)
59.         cout<<"左孩子:"<<b.leftChild(tmp)->data<<endl;
60.     else
61.         cout<<"无左孩子"<<endl;
62.     if(b.rightChild(tmp)!=NULL)
63.         cout<<"右孩子:"<<b.rightChild(tmp)->data<<endl;
64.     else
65.         cout<<"无右孩子"<<endl;
66.     cout<<endl;
67.
68.     //输出二叉树叶子结点个数
69.     cout<<"二叉树叶子结点个数: "<<b.getLeaves()<<endl<<endl;
70.
71.     //输出二叉树高度
72.     cout<<"二叉树高度: "<<b.getHeight();
73.
74.     //释放二叉树
75.     b.~binaryTree();
76. }

```

运行截图



实验内容题目四：线索化二叉树

3. 线索化二叉树的操作：

- (1) 编写一个程序实现二叉树的中序线索化，采用递归和非递归方式输出中序线索二叉树的中序序列，并以图（一）所示的二叉树 B 对程序进行验证；
- (2) 求出该线索化二叉树中给定结点的直接前驱和直接后继结点。

代码

binarytree.h

```

1. #ifndef BINARYTREE_H
2. #define BINARYTREE_H
3.
4. #include <iostream>
5.
6. struct binaryTreeNode{
7.     int leftTag=0,rightTag=0;
8.     char data; //元素
9.     binaryTreeNode *leftChild, *rightChild; //左孩子,右孩子
10.    binaryTreeNode(){leftChild = NULL; rightChild = NULL;}
11.    binaryTreeNode(char d){
12.        data = d;
13.        leftChild = NULL;
14.        rightChild = NULL;
15.    }
16. };
17.
18. class binaryTree
19. {
20. protected:
21.
22.     char end; //输入终止符
23.     binaryTreeNode *parent(binaryTreeNode *subTree,binaryTreeNode*d); //返回父结点
24.     int size(binaryTreeNode *subTree); //返回结点个数
25.     int height(binaryTreeNode *subTree); //返回高度
26.     void Insert(binaryTreeNode *subTree, char d); //插入
27.     void destory(binaryTreeNode *subTree); //删除
28.     bool find(binaryTreeNode *subTree, char d); //查找
29.
30. public:
31.     binaryTreeNode *root; //根
32.     binaryTree(){root = NULL;}
33.     binaryTree(char end){this->end = end; root = NULL;}
34.     ~binaryTree(){destory(root);}
35.
36.     void createTreeByPreOrder(binaryTreeNode * &subTree); //前序遍历创建树
37.     binaryTreeNode * createTreeByPreOrderAndInfixOrder
38.     (int preFront, int preRear, int infFront, int infRear, char preOrder[],
39.     char infixOrder[]); //根据先序中序创建树

```

```

40.     binaryTreeNode *parent(binaryTreeNode *d){
41.         if(root!=NULL && root!=d)
42.             return parent(root, d);
43.         else
44.             return NULL;
45.     } //返回父结点
46.     binaryTreeNode *leftChild(binaryTreeNode *d){
47.         if(d!=NULL)
48.             return d->leftChild;
49.         else
50.             return NULL;
51.     } //返回左孩子
52.     binaryTreeNode *rightChild(binaryTreeNode *d){
53.         if(d!=NULL)
54.             return d->rightChild;
55.         else
56.             return NULL;
57.     } //返回右孩子
58.
59.     bool isEmpty(){return root==NULL;} //判断是否为空
60.     binaryTreeNode *getRoot(){return root;} //返回根结点
61.     char getData(binaryTreeNode *tree){return tree->data;}
62.     int getSize(){return size(root);} //获取结点数
63.     int getHeight(){return height(root);} //获取高度
64.     binaryTreeNode *getNodeByData(char d);
65.
66.     //递归算法
67.     void preOrder_recursion(binaryTreeNode *subTree); //前序遍历
68.     void infixOrder_recursion(binaryTreeNode *subTree); //中序遍历
69.     void postOrder_recursion(binaryTreeNode *subTree); //后序遍历
70.
71.     //非递归算法
72.     void preOrder_NONrecursion(binaryTreeNode *subTree); //前序遍历
73.     void infixOrder_NONrecursion(binaryTreeNode *subTree); //中序遍历
74.
75.     void output(binaryTreeNode *b);
76.
77.     //中序线索化
78.     void createInfixThread();
79.     void createInfixThread(binaryTreeNode *d, binaryTreeNode *&pre);
80.
81.     //访问中序下第一个结点
82.     binaryTreeNode *first(binaryTreeNode *b){
83.         binaryTreeNode *bin = b;

```

```

84.     while(bin->leftTag == 0)
85.         bin = bin->leftChild;
86.     return bin;
87. }
88.
89. //访问中序下最后一个结点
90. binaryTreeNode *last(binaryTreeNode *b){
91.     binaryTreeNode *bin = b;
92.     while(bin->rightTag == 0)
93.         bin = bin->rightChild;
94.     return bin;
95. }
96.
97. //访问中序下前驱结点
98. binaryTreeNode *pre(binaryTreeNode *b){
99.     binaryTreeNode *bin = b->leftChild;
100.    if(b->leftTag == 0)
101.        return last(bin);
102.    else
103.        return bin;
104. }
105.
106. //访问中序下后继结点
107. binaryTreeNode *next(binaryTreeNode *b){
108.     binaryTreeNode *bin = b->rightChild;
109.     if(b->rightTag == 0)
110.         return first(bin);
111.     else
112.         return bin;
113. }
114. };
115.
116. #endif // BINARYTREE_H

```

binarytree.cpp

```

1. #include "binarytree.h"
2. #include "stack"
3.
4. void binaryTree::createTreeByPreOrder(binaryTreeNode * &subTree){
5.     char c;
6.     std::cin>>c;
7.     if(c!=end){
8.         //printf("%c",end);

```

```

9.         subTree = new binaryTreeNode(c);
10.        //subTree->data = c;
11.        createTreeByPreOrder(subTree->leftChild);
12.        createTreeByPreOrder(subTree->rightChild);
13.    }else{
14.        subTree = NULL;
15.    }
16. }
17.
18. binaryTreeNode * binaryTree::createTreeByPreOrderAndInfixOrder
19. (int preFront, int preRear, int infFront, int infRear, char preOrder[], char
    infixOrder[]){
20.     if(preFront>preRear||infFront>infRear)
21.         return NULL;
22.     binaryTreeNode *subTree = new binaryTreeNode;
23.     subTree->data = preOrder[preFront];    //根节点
24.     int flag=0;
25.     for(int i=infFront; i<=infRear; ++i){
26.         if(preOrder[preFront] == infixOrder[i]){
27.             flag = i;
28.             break;
29.         }
30.     }
31.     subTree->leftChild = createTreeByPreOrderAndInfixOrder(preFront+1,preFro
        nt+flag-infFront,infFront,flag-1,preOrder,infixOrder);
32.     subTree->rightChild = createTreeByPreOrderAndInfixOrder(preFront+flag-in
        fFront+1,preRear,flag+1,infRear,preOrder,infixOrder);
33.
34.     return subTree;
35. }
36.
37. void binaryTree::destory(binaryTreeNode *subTree){
38.     if(subTree != NULL){
39.         destory(subTree->leftChild);
40.         destory(subTree->rightChild);
41.         delete subTree;
42.     }
43. }
44.
45. binaryTreeNode * binaryTree::parent(binaryTreeNode *subTree, binaryTreeNode
    *d){
46.     //不存在则返回空
47.     if(subTree == NULL)
48.         return NULL;

```

```

49.
50.     //找到则返回
51.     if(subTree->leftChild == d || subTree->rightChild == d)
52.         return subTree;
53.
54.     //递归搜索
55.     binaryTreeNode *search = parent(subTree->leftChild);
56.     if(search != NULL)
57.         return search;
58.     else
59.         return parent(subTree->rightChild,d);
60. }
61.
62. //递归算法
63. void binaryTree::infixOrder_recursion(binaryTreeNode *subTree){
64.     if(subTree != NULL){
65.         if(subTree->leftTag == 0)
66.             infixOrder_recursion(subTree->leftChild);
67.         printf("%c",subTree->data);
68.         if(subTree->rightTag == 0)
69.             infixOrder_recursion(subTree->rightChild);
70.     }
71. }
72.
73. //非递归算法
74. void binaryTree::infixOrder_NONrecursion(binaryTreeNode *subTree){
75.     std::stack<binaryTreeNode *> s;
76.     binaryTreeNode * bin = root;
77.     do{
78.         while(bin != NULL){
79.             s.push(bin);
80.             if(bin->leftTag == 0)
81.                 bin = bin->leftChild;
82.             else
83.                 bin = NULL;
84.         }
85.         if(!s.empty()){
86.             bin = s.top();
87.             s.pop();
88.             printf("%c",bin->data);
89.             if(bin->rightTag == 0)
90.                 bin = bin->rightChild;
91.             else
92.                 bin = NULL;

```



```

93.     }
94.   }while(bin!=NULL || !s.empty());
95. }
96.
97. binaryTreeNode *binaryTree::getNodeByData(char d){
98.     std::stack<binaryTreeNode *> s;
99.     binaryTreeNode *bin = root;
100.    int flag = 0;
101.    s.push(NULL);
102.    while(bin!=NULL){
103.        if(bin->data == d){
104.            flag = 1;
105.            break;
106.        }
107.        if(bin->rightChild != NULL)
108.            s.push(bin->rightChild);
109.        if(bin->leftChild != NULL)
110.            bin = bin->leftChild;
111.        else{
112.            bin = s.top();
113.            s.pop();
114.        }
115.    }
116.    if(flag)
117.        return bin;
118.    else
119.        return NULL;
120. }
121.
122. int binaryTree::size(binaryTreeNode *subTree){
123.     if(subTree == NULL)
124.         return 0;
125.     else
126.         return size(subTree->leftChild)+size(subTree->rightChild)+1;
127. }
128.
129. int binaryTree::height(binaryTreeNode *subTree){
130.     if(subTree == NULL)
131.         return 0;
132.     else{
133.         int i = height(subTree->leftChild);
134.         int j = height(subTree->rightChild);
135.         return i<j?j+1:i+1;
136.     }

```

```

137. }
138.
139.
140.
141. void binaryTree::output(binaryTreeNode *b){
142.     if(b != NULL){
143.         std::cout<<b->data;
144.         if(b->leftChild!=NULL||b->rightChild!=NULL){
145.             std::cout<<'(';
146.             output(b->leftChild);
147.             std::cout<<",";
148.             if(b->rightChild!=NULL)
149.                 output(b->rightChild);
150.             std::cout<<")";
151.         }
152.     }
153. }
154.
155. void binaryTree::createInfixThread(){
156.     binaryTreeNode *pre = NULL; //前驱结点
157.     if(root != NULL){
158.         createInfixThread(root,pre);
159.         //处理最后一个结点
160.         pre->rightChild = NULL;
161.         pre->rightTag = 1;
162.     }
163. }
164.
165. void binaryTree::createInfixThread(binaryTreeNode *d, binaryTreeNode *&pre)
    {
166.     if(d == NULL)
167.         return;
168.     //递归使左子树线索化
169.     createInfixThread(d->leftChild,pre);
170.
171.     //建立前驱结点
172.     if(d->leftChild == NULL){
173.         d->leftChild = pre;
174.         d->leftTag = 1;
175.     }
176.
177.     //建立前驱结点的后继结点
178.     if(pre != NULL && pre->rightChild == NULL){
179.         pre->rightChild = d;

```

```

180.     pre->rightTag =1;
181. }
182.     pre = d;
183.     //递归使右子树线索化
184.     createInfixThread(d->rightChild,pre);
185. }

```

main.cpp

```

1. #include <iostream>
2. #include "binarytree.h"
3. using namespace std;
4.
5. int main()
6. {
7.     //拓展先序序列创建树
8.     char end = '*';
9.     binaryTree b(end);
10.    printf("请输入所要创建的树的前序遍历:\n");
11.    b.createTreeByPreOrder(b.root);
12.    cout<<"创建完成"<<endl<<endl;
13.
14.    //中序线索化
15.    cout<<"开始进行中序线索化..."<<endl;
16.    b.createInfixThread();
17.    cout<<"中序线索化完成"<<endl<<endl;
18.
19.    //输出
20.    //递归
21.    //中序
22.    printf("递归中序输出: ");
23.    b.infixOrder_recursion(b.getRoot());
24.    cout<<endl;
25.    //非递归
26.    //中序
27.    printf("非递归中序输出: ");
28.    b.infixOrder_NONrecursion(b.getRoot());
29.    cout<<endl<<endl;
30.
31.    //输出二叉树中指定结点值的左右孩子结点
32.    char data;
33.    cout<<"请输入要查找左右孩子的结点的结点值: ";
34.    cin>>data;
35.    binaryTreeNode *tmp = b.getNodeByData(data);

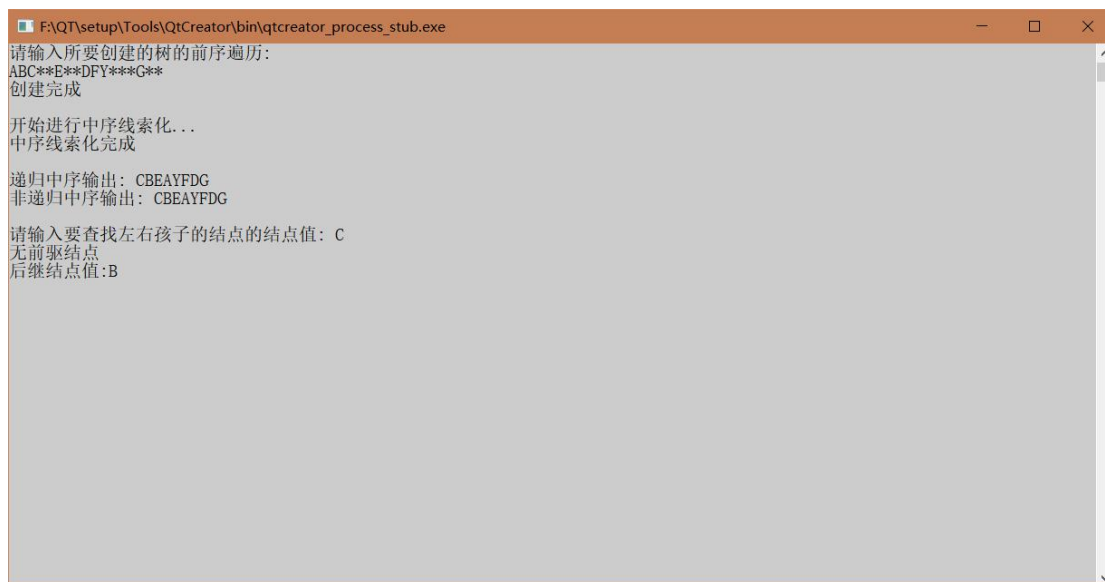
```

```

36.     if(b.pre(tmp)!=NULL)
37.         cout<<"前驱结点值:"<<b.pre(tmp)->data<<endl;
38.     else
39.         cout<<"无前驱结点"<<endl;
40.     if(b.next(tmp)!=NULL)
41.         cout<<"后继结点值:"<<b.next(tmp)->data<<endl;
42.     else
43.         cout<<"无后继结点"<<endl;
44.     cout<<endl;
45.
46.     //释放二叉树
47.     b.~binaryTree();
48. }

```

运行截图



实验五 图实验

【实验目的】

1. 掌握图的邻接矩阵、邻接表等存储表示方法
2. 掌握图的深度优先遍历和广度优先遍历算法
3. 了解最小生成树、最短路径等问题的求解过程、算法和应用
4. 了解关键路径、拓扑排序等问题的求解过程和应用

【实验学时】

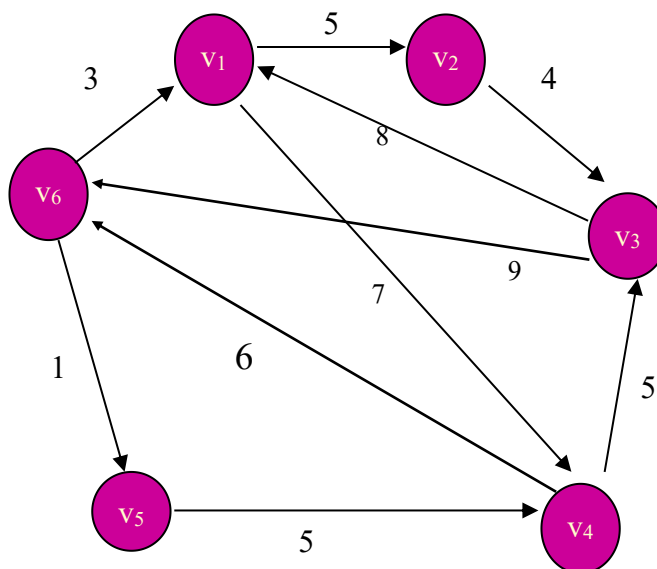
2 学时

【实验内容】

实验内容题目一：邻接矩阵-邻接表

题目一：编写一个程序，实现图的相关运算，并在此基础上设计一个主程序完成如下功能：

- (1) 建立图（二）所示的有向图 G 的邻接矩阵，并输出之。
- (2) 由有向图 G 的邻接矩阵产生邻接表，并输出之。
- (3) 再由（2）的邻接表产生对应的邻接矩阵，并输出之。



代码

```

1. #include <iostream>
2.
3. using namespace std;
4. const int n = 6; //点数
5.
6. //边结点
7. typedef struct arcNode{

```

```

8.     int indexNext;    //另一个顶点位置
9.     int w=0;    //权重
10.    arcNode *next = NULL;    //下一条边
11.    arcNode(){}
12.    arcNode(int num, int w):indexNext(num),w(w),next(NULL){}
13. }arcNode;
14.
15. //点结点
16. struct vNode{
17.     int num=0;    //顶点编号 v1 对应 0
18.     struct arcNode *head = NULL;    //边的头结点
19. };
20.
21. //邻接表
22. struct table{
23.     int numArc=0;    //边数
24.     vNode v[n];    //顶点头结点
25.     int arcNum;    //边数
26.     void insertV(int i, int n){
27.         v[i].num = n;
28.         v[i].head = NULL;
29.     }
30.     void insertArc(int v1, int v2, int w){
31.         if(v1>=0&&v1<n&&v2>=0&&v2<n){
32.             arcNode *q,*p = v[v1].head;
33.             q = new arcNode;
34.             //顺着链找合适的插入位置
35.             while(p!=NULL){
36.                 if(p->indexNext == v2){
37.                     cout<<"边已存在\n";
38.                     return;
39.                 }
40.                 q = p;
41.                 p = p->next;
42.             }
43.             p = new arcNode;
44.             q->next = p;
45.             if(v[v1].head == NULL)
46.                 v[v1].head = p;
47.             p->indexNext = v2;
48.             p->w = w;
49.             p->next = NULL;
50.             //cout<<"加入边成功! \n";
51.         }

```

```

52.     }
53.
54. };
55.
56. int main()
57. {
58.     // a[i][j] 表示 v(i+1) 到 v(j+1) 的权
59.     int a[n][n] = {0};
60.
61.     //创建邻接矩阵
62.     a[0][1] = 5;
63.     a[0][3] = 7;
64.     a[1][2] = 4;
65.     a[2][0] = 8;
66.     a[2][5] = 9;
67.     a[3][2] = 5;
68.     a[3][5] = 6;
69.     a[4][3] = 5;
70.     a[5][0] = 3;
71.     a[5][4] = 1;
72.
73.     //输出
74.     cout<<"邻接矩阵: "<<endl;
75.     for(int i=0; i<n; ++i){
76.         for(int j=0; j<n; ++j)
77.             cout<<a[i][j]<<" ";
78.         cout<<endl;
79.     }
80.     cout<<endl;
81.
82.     //创建邻接表
83.     table aTable;
84.     aTable.arcNum = 10; //设置边数
85.     //为顶点编号
86.     for(int i=0; i<n; ++i){
87.         aTable.insertV(i,i);
88.     }
89.
90.     //创建邻接表
91.     for(int i=0; i<n; ++i){
92.         for(int j=0; j<n; ++j){
93.             if(a[i][j] != 0){
94.                 aTable.insertArc(i,j,a[i][j]);
95.             }

```

```

96.     }
97. }
98. cout<<"邻接表创建完成"<<endl<<endl;
99.
100. //输出
101. cout<<"邻接表: \n";
102. for(int i=0; i<n; ++i){
103.     cout<<"v"<<i+1<<": ";
104.     arcNode *tmp = aTable.v[i].head; //记录头结点
105.     while(tmp!=NULL){
106.         cout<<"w:"<<tmp->w<<" ";
107.         cout<<"v"<<tmp->indexNext+1<<" ";
108.         tmp = tmp->next;
109.     }
110.     cout<<endl;
111. }
112.
113. //创建邻接矩阵
114. int b[n][n] = {0};
115. for(int i=0; i<n; ++i){
116.     arcNode *tmp = aTable.v[i].head; //记录头结点
117.     while(tmp!=NULL){
118.         b[i][tmp->indexNext] = tmp->w;
119.         tmp = tmp->next;
120.     }
121. }
122. cout<<endl;
123. cout<<"邻接矩阵创建完成"<<endl<<endl;
124.
125. //输出
126. cout<<"邻接矩阵: "<<endl;
127. for(int i=0; i<n; ++i){
128.     for(int j=0; j<n; ++j)
129.         cout<<b[i][j]<<" ";
130.     cout<<endl;
131. }
132. cout<<endl;
133.
134.
135.
136.
137. }

```


运行截图

```

邻接矩阵:
0 5 0 7 0 0
0 0 4 0 0 0
8 0 0 0 0 9
0 0 5 0 0 6
0 0 0 5 0 0
3 0 0 0 1 0

邻接表创建完成

邻接表:
v1: w:5 v2 w:7 v4
v2: w:4 v3
v3: w:8 v1 w:9 v6
v4: w:5 v3 w:6 v6
v5: w:5 v4
v6: w:3 v1 w:1 v5

邻接矩阵创建完成

邻接矩阵:
0 5 0 7 0 0
0 0 4 0 0 0
8 0 0 0 0 9
0 0 5 0 0 6
0 0 0 5 0 0
3 0 0 0 1 0
    
```

实验内容题目二：图的遍历

题目二：编写一个程序，实现图的遍历算法，并在此基础上设计一个主程序完成如下功能：

- (1) 输出图（二）的有向图 G 从顶点 V1 开始的深度优先遍历序列序列（递归算法）
- (2) 输出如图（二）的有向图 G 从顶点 v1 开始的深度优先遍历序列（非递归算法）
- (3) 输出如图（二）的有向图 G 从顶点 v1 开始的广度优先遍历序列（选做）

代码

```

1. #include <iostream>
2. #include "stack"
3. using namespace std;
4. const int n = 6; //点数
5. bool visit[n] = {0}; //标记是否被访问
6.
7. //边结点
8. typedef struct arcNode{
9.     int indexNext; //另一个顶点位置
10.    int w=0; //权重
11.    arcNode *next = NULL; //下一条边
12.    arcNode(){}
13.    arcNode(int num, int w):indexNext(num),w(w),next(NULL){}
14. }arcNode;
15.
16. //点结点
17. struct vNode{
18.    int num=0; //顶点编号 v1 对应 0
    
```

```

19.     struct arcNode *head = NULL; //边的头结点
20. };
21.
22. //邻接表
23. struct table{
24.     int numArc=0; //边数
25.     vNode v[n]; //顶点头结点
26.     int arcNum; //边数
27.     void insertV(int i, int n){
28.         v[i].num = n;
29.         v[i].head = NULL;
30.     }
31.     void insertArc(int v1, int v2, int w){
32.         if(v1>=0&&v1<n&&v2>=0&&v2<n){
33.             arcNode *q,*p = v[v1].head;
34.             q = new arcNode;
35.             //顺着链找合适的插入位置
36.             while(p!=NULL){
37.                 if(p->indexNext == v2){
38.                     cout<<"边已存在\n";
39.                     return;
40.                 }
41.                 q = p;
42.                 p = p->next;
43.             }
44.             p = new arcNode;
45.             q->next = p;
46.             if(v[v1].head == NULL)
47.                 v[v1].head = p;
48.             p->indexNext = v2;
49.             p->w = w;
50.             p->next = NULL;
51.             //cout<<"加入边成功! \n";
52.         }
53.     }
54.
55. };
56.
57. //visit[] 清0
58. void makeVisit_0(){
59.     for(int i=0; i<n; ++i)
60.         visit[i] = 0;
61. }
62.

```

```

63. //深度优先算法遍历 DFS 递归
64. void DFS_recursion(table G, int i){
65.     arcNode *p;
66.     cout<<"v"<<G.v[i].num+1<<" ";
67.     visit[i] = 1;
68.     for(p=G.v[i].head; p!=NULL; p=p->next){
69.         if(!visit[p->indexNext])
70.             DFS_recursion(G,p->indexNext);
71.     }
72. }
73.
74. //深度优先算法遍历 DFS 非递归
75. void DFS_NONrecursion(table G, int i){
76.     stack<arcNode *> s;
77.     int size = 0;
78.     arcNode *p = G.v[i].head;
79.
80.     cout<<"v"<<G.v[i].num+1<<" ";
81.     visit[i] = 1;    //已访问
82.     size++;
83.
84.     s.push(p);
85.     int j=0;
86.
87.     while(size<6){
88.         bool flag = 1;
89.         while(p->indexNext<6&&p->indexNext>=0){
90.             if(visit[p->indexNext]==0){
91.                 cout<<"v"<<G.v[p->indexNext].num+1<<" ";
92.                 visit[p->indexNext] = 1;
93.                 p = G.v[p->indexNext].head;
94.                 size++;
95.                 flag = 0;
96.                 break;
97.             }
98.             p = p->next;
99.         }
100.        if(flag){
101.            p = s.top();
102.            s.pop();
103.        }
104.    }
105. }
106.

```

```

107. //广度优先遍历 BFS
108. void BFS(table G, int i){
109.     for(int j=0; j<n; ++i){
110.         if(visit[j]==0){
111.             cout<<"v"<<j+1<<" ";
112.             visit[j] = 1;
113.         }
114.         arcNode *tmp = G.v[i].head; //记录头结点
115.         while(tmp!=NULL){
116.             if(visit[tmp->indexNext]==0){
117.                 cout<<"v"<<tmp->indexNext+1<<" ";
118.                 visit[tmp->indexNext] = 1;
119.             }
120.             tmp = tmp->next;
121.         }
122.     }
123. }
124.
125. int main()
126. {
127.     // a[i][j] 表示 v(i+1) 到 v(j+1) 的权
128.     int a[n][n] = {0};
129.
130.     //创建邻接矩阵
131.     a[0][1] = 5;
132.     a[0][3] = 7;
133.     a[1][2] = 4;
134.     a[2][0] = 8;
135.     a[2][5] = 9;
136.     a[3][2] = 5;
137.     a[3][5] = 6;
138.     a[4][3] = 5;
139.     a[5][0] = 3;
140.     a[5][4] = 1;
141.
142.     //输出
143.     cout<<"邻接矩阵: "<<endl;
144.     for(int i=0; i<n; ++i){
145.         for(int j=0; j<n; ++j)
146.             cout<<a[i][j]<<" ";
147.         cout<<endl;
148.     }
149.     cout<<endl;
150.

```

```

151.    //创建邻接表
152.    table aTable;
153.    aTable.arcNum = 10; //设置边数
154.    //为顶点编号
155.    for(int i=0; i<n; ++i){
156.        aTable.insertV(i,i);
157.    }
158.
159.    //创建邻接表
160.    for(int i=0; i<n; ++i){
161.        for(int j=0; j<n; ++j){
162.            if(a[i][j] != 0){
163.                aTable.insertArc(i,j,a[i][j]);
164.            }
165.        }
166.    }
167.    cout<<"邻接表创建完成"<<endl<<endl;
168.
169.    //输出
170.    cout<<"邻接表: \n";
171.    for(int i=0; i<n; ++i){
172.        cout<<"v"<<i+1<<":  ";
173.        arcNode *tmp = aTable.v[i].head; //记录头结点
174.        while(tmp!=NULL){
175.            cout<<"w:"<<tmp->w<<" ";
176.            cout<<"v"<<tmp->indexNext+1<<" ";
177.            tmp = tmp->next;
178.        }
179.        cout<<endl;
180.    }
181.    cout<<endl;
182.
183.    //从顶点 v1 开始的深度优先遍历序列序列（递归算法）
184.    makeVisit_0();
185.    cout<<"从顶点 v1 开始的深度优先遍历序列序列（递归算法）: \n";
186.    DFS_recursion(aTable,0);
187.    cout<<endl;
188.
189.    //从顶点 v1 开始的深度优先遍历序列序列（非递归算法）
190.    makeVisit_0();
191.    cout<<"从顶点 v1 开始的深度优先遍历序列序列（非递归算法）: \n";
192.    DFS_NONrecursion(aTable,0);
193.    cout<<endl;
194.

```

```

195.    //从顶点 v1 开始的广度优先遍历序列
196.    makeVisit_0();
197.    cout<<"从顶点 v1 开始的广度优先遍历序列：\n";
198.    BFS(aTable,0);
199.    cout<<endl;
200.
201.
202.
203.
204.
205.
206. }
    
```

运行截图

```

F:\QT\setup\Tools\QtCreator\bin\qtcreator_process_stub.exe
邻接矩阵:
0 5 0 7 0 0
0 0 4 0 0 0
8 0 0 0 0 9
0 0 5 0 0 6
0 0 0 5 0 0
3 0 0 0 1 0

邻接表创建完成

邻接表:
v1:  w:5 v2  w:7 v4
v2:  w:4 v3
v3:  w:8 v1  w:9 v6
v4:  w:5 v3  w:6 v6
v5:  w:5 v4
v6:  w:3 v1  w:1 v5

从顶点v1开始的深度优先遍历序列序列（递归算法）:
v1 v2 v3 v6 v5 v4
从顶点v1开始的深度优先遍历序列序列（非递归算法）:
v1 v2 v3 v6 v5 v4
从顶点v1开始的广度优先遍历序列:
v1 v2 v4 v3 v6 v5
    
```

【实训项目】子工程建设时间的安排（选做？）

实训项目题目一：子工程建设时间的安排

现某公司有一计算机机房建设工程，它包含的子工程以及各子工程之间的关系如表 2 所示。

由于资金和场地等条件限制，这些子工程必须一项一项地进行，不能有并行情况。请给出一种可行的安排这些子工程建设时间的一个线性序列，按照它的顺序依次进行各个子工程的建设，以顺利完成整个工程。

表 2 计算机机房建设工程表

子工程代号	子工程名称	前序子工程
V1	设计图纸	无
V2	招标公司	无
V3	机房装修	V1 V2
V4	购买电源	V1
V5	购买机器	V2
V6	装机组网	V3 V4 V5
V7	软件配置	V6
V8	网络调试	V6 V7
V9	工程验收	V6 V7 V8

代码

```

1. #include <iostream>
2. using namespace std;
3. const int n=9;
4.
5. bool isAllVisited(int visit[]){
6.     for(int i=0; i<n; ++i)
7.         if(visit[i]==0)
8.             return false;
9.     return true;
10. }
11.
12. int main()
13. {
14.     int time[n];    //访问次序
15.     int visit[n] = {0}; //是否访问
16.     int v[n] = {0,0,2,1,1,3,1,2,3}; //入度
17.     int j=0;
18.     while(!isAllVisited(visit)){
19.         for(int i=0; i<n; ++i){
20.             if(v[i]==0 && visit[i]==0){
21.                 time[j]=i;
22.                 j++;
23.                 visit[i] = 1;
24.                 //遇到如下点 则相应入度减少
25.                 switch (i) {
26.                     case 0:{
27.                         v[2]--;

```

```

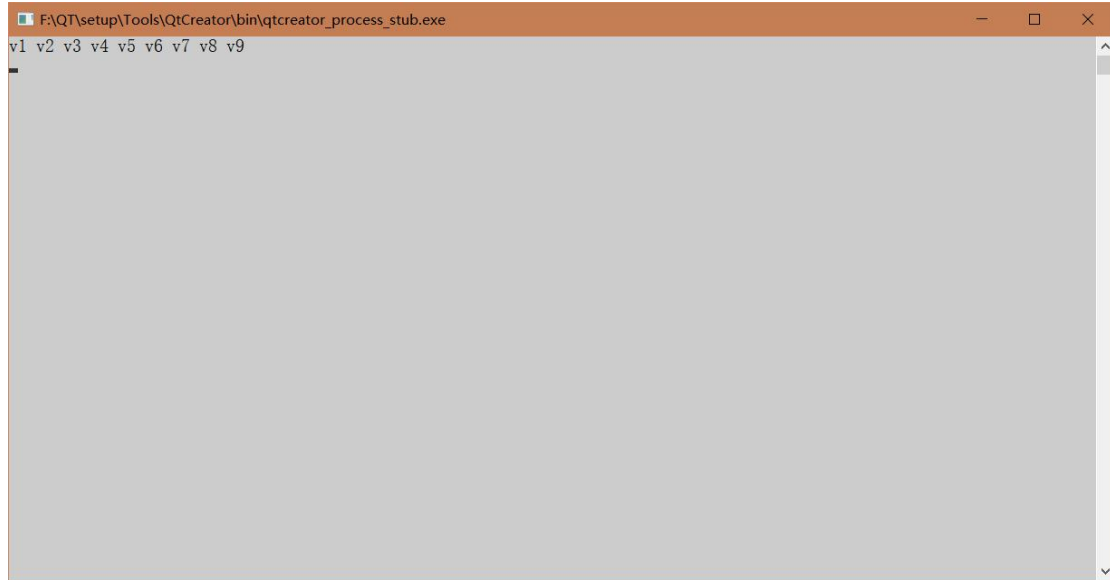
28.         v[3]--;
29.         break;
30.     }
31.     case 1:{
32.         v[2]--;
33.         v[4]--;
34.         break;
35.     }
36.     case 2:{
37.         v[5]--;
38.         break;
39.     }
40.     case 3:{
41.         v[5]--;
42.         break;
43.     }
44.     case 4:{
45.         v[5]--;
46.         break;
47.     }
48.     case 5:{
49.         v[6]--;
50.         v[7]--;
51.         v[8]--;
52.         break;
53.     }
54.     case 6:{
55.         v[7]--;
56.         v[8]--;
57.         break;
58.     }
59.     case 7:{
60.         v[8]--;
61.         break;
62.     }
63.     default:
64.         break;
65.     }
66. }else
67.     i++;
68. }
69. }
70. for(int i=0; i<n; ++i)
71.     cout<<"v"<<time[i]+1<<" ";

```



```
72.     cout<<endl;  
73. }
```

运行截图



实验六 查找表实验

【实验目的】

1. 掌握顺序查找、二分查找等算法的思想、查找过程、算法实现及应用
2. 了解哈希表的构造、哈希函数的构造方法及处理冲突的方法
3. 了解二叉排序树的构造、查找等算法

【实验学时】

2 学时

【实验内容】

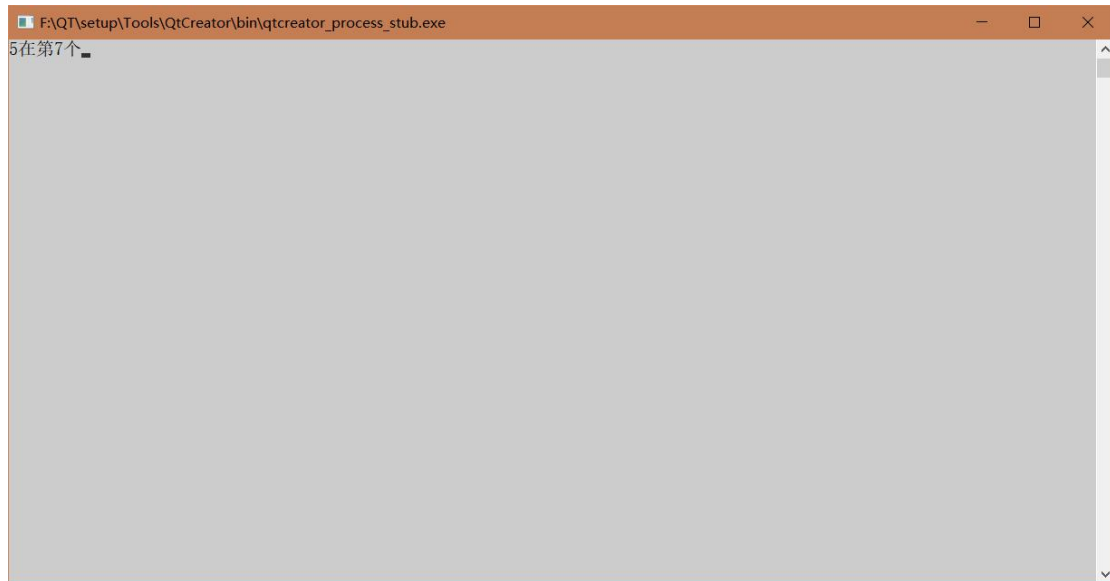
实验内容题目一：顺序查找

题目一：编写一个程序，输出在顺序表 {3, 6, 2, 10, 1, 8, 5, 7, 4, 9} 中采用顺序方法查找关键字 5 的过程。

代码

```
1. #include <iostream>
2.
3. using namespace std;
4.
5. //顺序查找
6. //寻找 x 所在位置，返回下标
7. int search(int a[],int length,int x){
8.     for(int i=0; i<length; ++i)
9.         if(a[i]==x)
10.            return i;
11. }
12.
13. int main()
14. {
15.     int a[10] = {3,6,2,10,1,8,5,7,4,9};
16.     int x = 5;
17.     int location = search(a,10,x)+1;
18.     printf("%d 在第%d 个",x,location);
19. }
```

运行截图



实验内容题目二：二分查找

题目二：编写一个程序，输出在顺序表 {1, 2, 3, 4, 5, 6, 7, 8, 9, 10} 中采用二分查找方法查找关键字 5 的过程。

代码

```

1. #include <iostream>
2.
3. using namespace std;
4.
5. //二分法 返回下标
6. int dichotomy(int a[],int length,int x){
7.     int front=0, rear=length-1;
8.     int mid = (rear+front)/2;
9.     while(front <= rear){
10.        if(a[mid] > x){
11.            rear = mid+1;
12.        }else if(a[mid] < x){
13.            front = mid-1;
14.        }else{
15.            return mid;
16.        }
17.    }
18. }
19.
20. int main()
21. {

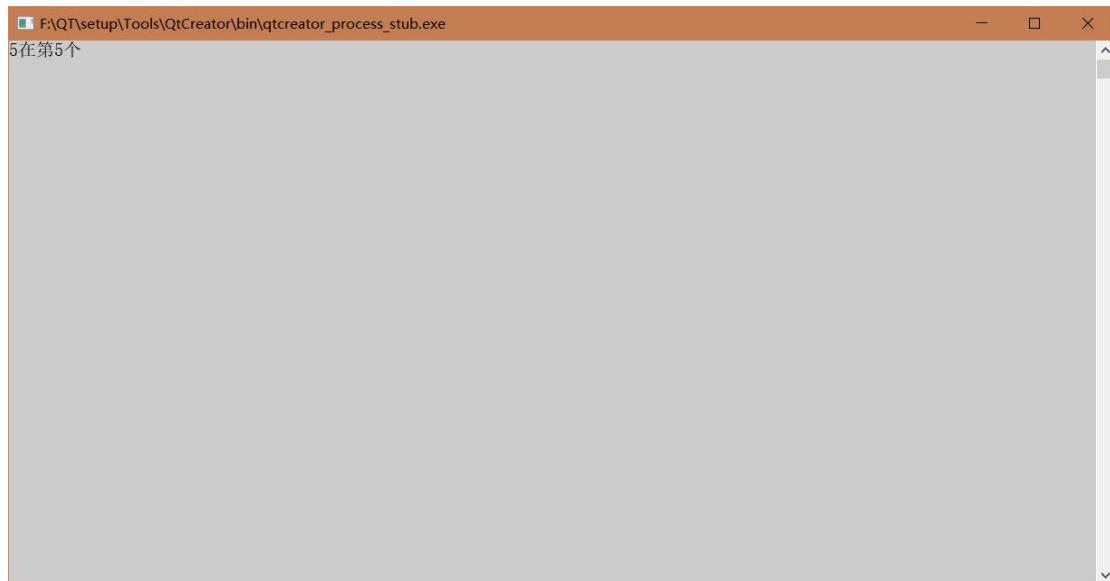
```

```

22.     int a[10] = {1,2,3,4,5,6,7,8,9,10};
23.     int x = 5;
24.     int location = dichotomy(a,10,x)+1;
25.     printf("%d 在第%d 个",x,location);
26.
27. }

```

运行截图



实验内容题目三：哈希查找

题目三：编写一个程序实现哈希表的相关运算，并在此基础上完成如下功能：

- (1) 建立 {16, 74, 60, 43, 54, 90, 46, 31, 29, 88, 77} 哈希表 $A[0 \cdots 12]$ ，哈希函数为： $H(k) = \text{key} \% p$ ，并采用线性探测再散列法解决冲突
- (2) 在上述哈希表中查找关键字为 29 的记录
- (3) 在上述哈希表中删除关键字为 77 的记录，再将其插入。

代码

```

1. #include <iostream>
2. using namespace std;
3. const int p = 13;
4.
5. int H(int key){
6.     return key%p;
7. }
8.
9. //存入哈希表

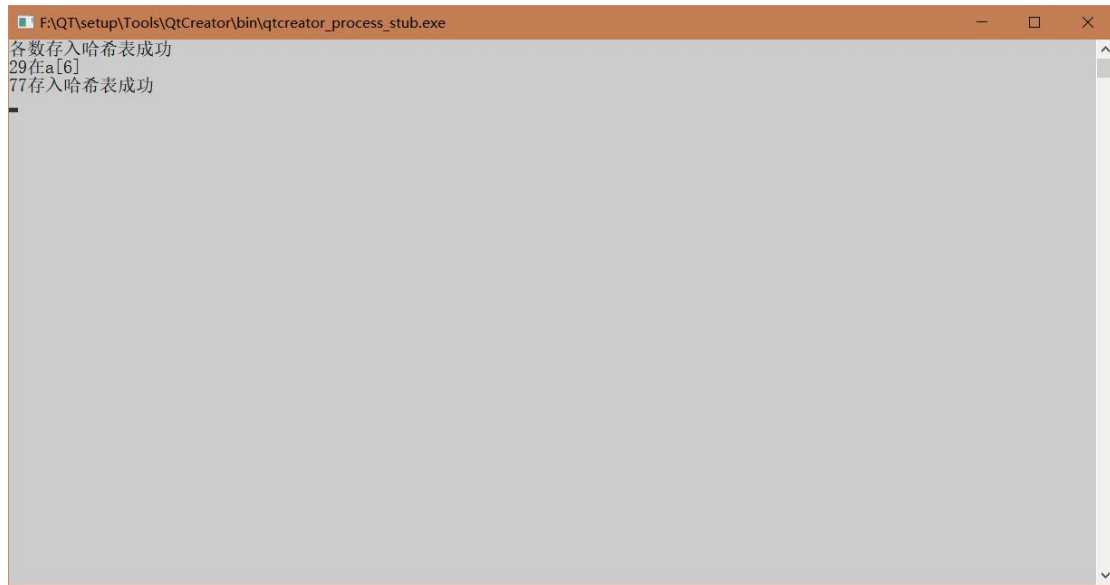
```

```

10. void HashTable(int a[], int key){
11.     int index = H(key);
12.     while(a[index]!=0){
13.         index++;
14.         if(index==12)
15.             index = 0;
16.     }
17.     a[index]=key;
18. }
19.
20. //查找 返回下标
21. int search(int a[], int d){
22.     int index = H(d);
23.     while(a[index]!=d){
24.         index++;
25.         if(index==12)
26.             index = 0;
27.     }
28.     return index;
29. }
30.
31. int main()
32. {
33.     int a[12] = {0};    //0 记为空
34.     for(int i=0; i<12; ++i)
35.         a[i]=0;
36.     int b[11] = {16,74,60,43,54,90,46,31,29,88,77};
37.
38.     //存入哈希表
39.     for(int i=0; i<11; ++i){
40.         HashTable(a,b[i]);
41.     }
42.     cout<<"各数存入哈希表成功"<<endl;
43.
44.     //查找关键字为 29 的记录
45.     int index29 = search(a,29);
46.     cout<<"29 在 a["<<index29<<"]"<<endl;
47.
48.     //删除 77，再将其插入
49.     int index77 = search(a,77);
50.     a[index77] = 0;
51.     HashTable(a,77);
52.     cout<<"77 存入哈希表成功"<<endl;
53. }

```

运行截图



实验七 内排序实验

【实验目的】

1. 掌握直接插入排序、冒泡排序、快速排序、希尔排序等排序方法的基本思想、排序过程、算法实现及其稳定性、复杂度和适用场合等特性及应用
2. 了解堆排序、归并排序、基数排序等排序方法的基本思想、排序过程、稳定性、复杂度和适用场合等特性

【实验学时】

2 学时

【实验内容】

实验内容题目一：直接插入

题目一：编写一个程序实现直接插入排序过程，并输出 {9, 8, 7, 6, 5, 4, 3, 2, 1, 0} 的排序过程。

代码

```

1. #include <iostream>
2. using namespace std;
3.
4. //直接插入排序法
5. void insertSort(int a[],int n){
6.     int flag = 0;
7.     for(int i=1; i<n; ++i){
8.         int j = i-1;
9.         while(a[i]<a[j]&&j!=-1){
10.            j--;
11.        }
12.        j++;
13.        int tmp = a[i];
14.        for(int k=i-1; k>=j; --k){
15.            a[k+1] = a[k];
16.        }
17.        a[j] = tmp;
18.        flag++;
19.        printf("第%d 次排序, 序列为: ",flag);
20.        for(int m=0; m<n; ++m)
21.            cout<<a[m]<<" ";
22.        cout<<endl;
23.    }

```

```

24. }
25.
26. int main()
27. {
28.     int a[11] = {9,8,7,6,5,4,3,2,1,0};
29.     insertSort(a,10);
30. }
    
```

运行截图

```

第1次排序, 序列为: 8 9 7 6 5 4 3 2 1 0
第2次排序, 序列为: 7 8 9 6 5 4 3 2 1 0
第3次排序, 序列为: 6 7 8 9 5 4 3 2 1 0
第4次排序, 序列为: 5 6 7 8 9 4 3 2 1 0
第5次排序, 序列为: 4 5 6 7 8 9 3 2 1 0
第6次排序, 序列为: 3 4 5 6 7 8 9 2 1 0
第7次排序, 序列为: 2 3 4 5 6 7 8 9 1 0
第8次排序, 序列为: 1 2 3 4 5 6 7 8 9 0
第9次排序, 序列为: 0 1 2 3 4 5 6 7 8 9
    
```

实验内容题目二：希尔排序

题目二：编写一个程序实现希尔排序过程，并输出 {9, 8, 7, 6, 5, 4, 3, 2, 1, 0} 的排序过程。

代码

```

1. #include <iostream>
2.
3. using namespace std;
4.
5. //直接插入排序法
6. void insertSort(int a[], int k){
7.     for(int i=1; i<k; ++i){
8.         int j = i-1;
9.         while(a[i]<a[j]&& j!=-1){
10.             j--;
11.         }
12.         j++;
    
```



```

13.     int tmp = a[i];
14.     for(int k=i-1; k>=j; --k){
15.         a[k+1] = a[k];
16.     }
17.     a[j] = tmp;
18. }
19. }
20.
21. //希尔排序
22. void shellSort(int a[],int n){
23.     int gap = n;
24.     int flag = 0;    //次数
25.     //int NN = n/3+1;
26.     do{
27.         gap = gap/3+1;    //间隔
28.         for(int i=0; i<gap; ++i){
29.             //int b[NN]={0};    //存放分组后的数组
30.             int b[10]={0};    //存放分组后的数组
31.             int k=0;
32.             //分组
33.             for(int j=i; j<n; j+=gap,k++)
34.                 b[k]=a[j];
35.             //对分组后的序列进行排序
36.             insertSort(b,k);
37.             k=0;
38.             for(int j=i; j<n; j+=gap,k++)
39.                 a[j]=b[k];
40.         }
41.         flag++;
42.         printf("第%d 次排序, 序列为: ",flag);
43.         for(int m=0; m<n; ++m)
44.             cout<<a[m]<<" ";
45.         cout<<endl;
46.     }while(gap>1);
47. }
48.
49. int main()
50. {
51.     int a[10] = {9,8,7,6,5,4,3,2,1,0};
52.     shellSort(a,10);
53. }

```

运行截图

```

F:\QT\setup\Tools\QtCreator\bin\qtcreator_process_stub.exe
第1次排序, 序列为: 1 0 3 2 5 4 7 6 9 8
第2次排序, 序列为: 1 0 3 2 5 4 7 6 9 8
第3次排序, 序列为: 0 1 2 3 4 5 6 7 8 9

```

实验内容题目三：快速排序

题目三：编写一个程序实现快速排序过程，并输出 {6, 8, 7, 9, 0, 1, 3, 2, 4, 5} 的排序过程。

代码

```

1. #include <iostream>
2.
3. using namespace std;
4. int flag=0;
5.
6. //调整序列，大的放右边 小的放左边
7. int adjust(int a[], int front, int rear){
8.     int i=front, j=rear;
9.     int x = a[front];
10.    while(i<j){
11.        while(a[j]>=x && i<j)
12.            j--;
13.        a[i] = a[j];
14.        while(a[i]<=x && i<j)
15.            i++;
16.        a[j] = a[i];
17.    }
18.    a[i] = x;
19.    return i;
20. }
21.
22. //快速排序

```

```

23. void quickSort(int a[],int front, int rear){
24.     if(front<rear){
25.         int empty = adjust(a,front,rear);
26.         flag++;
27.         printf("第%d 次排序, 序列为: ",flag);
28.         for(int m=0; m<10; ++m)
29.             cout<<a[m]<<" ";
30.         cout<<endl;
31.         quickSort(a,front,empty-1);
32.         quickSort(a,empty+1,rear);
33.     }
34. }
35.
36. int main()
37. {
38.     int a[11] = {9,8,7,6,5,4,3,2,1,0};
39.     quickSort(a,0,9);
40. }
    
```

运行截图

```

F:\QT\setup\Tools\QtCreator\bin\qtcreator_process_stub.exe
第1次排序, 序列为: 0 8 7 6 5 4 3 2 1 9
第2次排序, 序列为: 0 8 7 6 5 4 3 2 1 9
第3次排序, 序列为: 0 1 7 6 5 4 3 2 8 9
第4次排序, 序列为: 0 1 7 6 5 4 3 2 8 9
第5次排序, 序列为: 0 1 2 6 5 4 3 7 8 9
第6次排序, 序列为: 0 1 2 6 5 4 3 7 8 9
第7次排序, 序列为: 0 1 2 3 5 4 6 7 8 9
第8次排序, 序列为: 0 1 2 3 5 4 6 7 8 9
第9次排序, 序列为: 0 1 2 3 4 5 6 7 8 9
    
```