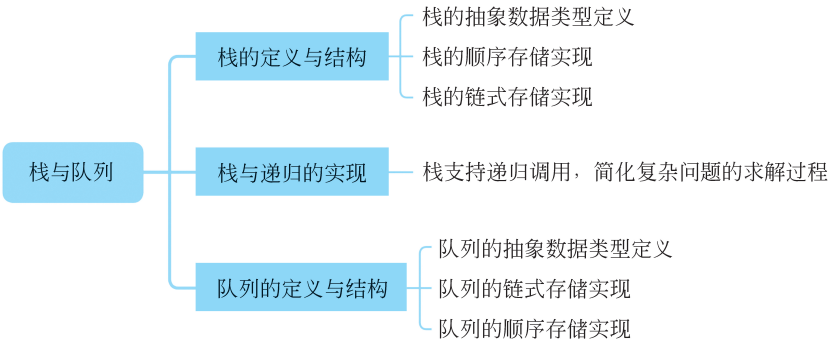
第 3 章　栈 与 队 列

3.1 导学

3.1.1 思维导图



3.1.2 重难点分析

1. 重点

栈的操作特性；顺序栈及实现；链栈及实现；队列的操作特性；循环队列及实现；链队列及实现。

2. 难点

两栈共享空间；队列的操作特性；循环队列及实现；链队列及实现。

3.2 案例

针对章节知识点进行案例讲解，结合教材，适合同学自学和教师课堂教学。

案例 3.1　数制转换

1. 实验目的

深入理解栈的“后进先出”特性，熟练掌握栈的顺序结构定义和使用。

2. 实验内容

利用栈完成数制转换，能够完成十进制向二进制、八进制、十六进制的转换。

3. 实验要求

栈可以采用顺序存储结构来存放栈内元素，并用一变量（如 top）始终指向栈顶元素以反映栈中元素的变化。

4. 实验分析

将十进制数 2025 转换为八进制数的过程如下：

N	$N \div d$	$N \bmod d$
2025	253	1
253	31	5
31	3	7
3	0	3

因此，$(2025)_{10}=(3751)_8$。在编程实现时，可以利用栈的特性：将计算过程中产生的八进制数依次入栈，最后按出栈顺序打印即可得到正确的结果。

5. 参考代码

```
#include <iostream>
#include <stack>
#include <string>
#include <stdexcept>

using namespace std;

//数制转换函数
string convertDecimalToBase(int decimalNumber, int base) {
    //检查基数是否合法
    if (base != 2 && base != 8 && base != 16) {
        throw invalid_argument("错误：只支持二进制(2)、八进制(8)和十六进制(16)");
    }

    //处理 0 的特殊情况
    if (decimalNumber == 0) {
        return "0";
    }

    stack<char> remainderStack;
    string result;

    //处理负数
    bool isNegative = false;
    if (decimalNumber < 0) {
        isNegative = true;
        decimalNumber = -decimalNumber; //转换为正数处理
    }

    //进行转换
    while (decimalNumber > 0) {
        int remainder = decimalNumber %base;

        //将余数转换为对应的字符
        if (remainder < 10) {
            remainderStack.push(static_cast<char>('0' + remainder));
        } else {
            //处理十六进制的 A-F
```

```
            remainderStack.push(static_cast<char>('A' + remainder - 10));
        }

        decimalNumber = decimalNumber / base;
    }

    //从栈中取出结果
    while (!remainderStack.empty()) {
        result += remainderStack.top();
        remainderStack.pop();
    }

    //如果是负数,添加负号
    if (isNegative) {
        result = "-" + result;
    }

    return result;
}

int main() {
    try {
        int decimalNumber;
        int base;

        cout << "===== 数制转换程序 =====" << endl;
        cout << "请输入一个十进制整数: ";
        cin >> decimalNumber;

        cout << "请选择目标进制 (2-二进制, 8-八进制, 16-十六进制): ";
        cin >> base;

        //验证输入
        if (cin.fail()) {
            throw invalid_argument("错误: 请输入有效的整数");
        }

        string result = convertDecimalToBase(decimalNumber, base);

        cout << endl;
        cout << "转换结果:" << endl;
        cout << decimalNumber << " (十进制) = " << result << " (";

        switch (base) {
            case 2:
                cout << "二进制)";
                break;
            case 8:
                cout << "八进制)";
                break;
            case 16:
                cout << "十六进制)";
                break;
```

```
        }

        cout << endl;

    } catch (const exception& e) {
        cout << endl << "错误: " << e.what() << endl;
        return 1;
    }

    return 0;
}
```

案例 3.2　括号匹配

1. 实验目的

深入理解栈的“后进先出”特性，熟练掌握栈的顺序结构定义和使用。

2. 实验内容

给定一个只包括(,),{,},[,]的字符串，判断字符串是否有效。有效字符串需满足：

(1) 左括号必须用相同类型的右括号闭合。

(2) 左括号必须以正确的顺序闭合。

注意：空字符串可被认为是有效字符串。

3. 实验要求

输入：

```
()[]{}
```

输出：

```
true
```

4. 实验分析

用栈这个数据结构来解决这个问题：依次扫描字符串中的括号，遇到左括号就入栈，遇到右括号就“消耗”(出栈)栈顶的左括号使之与右括号进行配对。

5. 参考代码

```
#include <iostream>
#include <stack>
#include <string>
using namespace std;

bool isValid(string s) {
    stack<char> bracketStack;

    //遍历字符串中的每个字符
    for (char c : s) {
        //如果是左括号，入栈
        if (c == '(' || c == '{' || c == '[') {
            bracketStack.push(c);
        }
        //如果是右括号
```

```
            else {
                //栈为空却遇到右括号,无效
                if (bracketStack.empty()) {
                    return false;
                }

                //取出栈顶元素,检查是否匹配
                char top = bracketStack.top();
                bracketStack.pop();

                if ((c == ')' && top != '(') ||
                    (c == '}' && top != '{') ||
                    (c == ']' && top != '[')) {
                    return false;
                }
            }
        }

        //所有字符处理完后,栈必须为空才有效
        return bracketStack.empty();
    }

    int main() {
        string s;
        cout << "请输入括号字符串: ";
        cin >> s;

        if (isValid(s)) {
            cout << "有效的括号字符串" << endl;
        } else {
            cout << "无效的括号字符串" << endl;
        }

        return 0;
    }
```

案例 3.3　行编辑程序

1. 实验目的

深入理解栈的“后进先出”特性,熟练掌握栈的定义和使用。

2. 实验内容

一个简单的行编辑程序的功能是：接收用户从终端输入的程序或数据,并存入用户的数据区。由于用户在终端上输入时,不能保证不出差错,因此,若在编辑行中,“每接收一个字符,即存入用户数据区”的做法显然是不恰当的。较好的做法是,设立一个输入缓冲区,用于接收用户输入的一行字符,然后逐行存入用户数据区。允许用户输入出差错,并在发现有误时,及时更正。例如,当用户发现刚刚输入的一个字符是错误的时候,可以补一个退格符“#”,以表示前一个字符无效;如果发现当前输入的行内差错较多的话,则可以输入一个退格符“@”,以表示当前行中的字符均无效。

3. 实验要求

输入：

```
whli##ilr#e(s#* s)
```

输出：

```
while (* s)
```

输入：

```
outcha@putchar(s = #+ + ) ;
```

输出：

```
putchar( * s ++ );
```

4. 实验分析

为了处理错误的输入，可以利用栈先进后出的特性，当读入一个字符时，如果这个字符不是＃或者@，则将该字符入栈；如果是＃则执行退格功能，即将栈顶元素出栈；如果当前读入的元素是@，则将栈清空，理解了这一过程，利用栈的基本操作，可以很容易地实现这一功能。

5. 参考代码

```
#include <iostream>
#include <stack>
#include <string>
using namespace std;

//处理行编辑的函数
string processLine(const string& input) {
    stack<char> buffer;                    //使用栈作为输入缓冲区

    for (char c : input) {
        if (c == '#') {                    //退格符：删除前一个字符
            if (!buffer.empty()) {
                buffer.pop();
            }
        }
        else if (c == '@') {               //清空符：删除当前行所有字符
            while (!buffer.empty()) {
                buffer.pop();
            }
        }
        else {                             //普通字符：直接入栈
            buffer.push(c);
        }
    }

    //将栈中字符转换为字符串(注意栈的逆序特性)
    string result;
    while (!buffer.empty()) {
        result = buffer.top() + result;    //每次在前面添加字符
        buffer.pop();
    }

    return result;
}
```

```
int main() {
    string input;
    cout << "请输入编辑行: ";
    getline(cin, input);        //使用 getline()获取包含空格的完整输入

    string output = processLine(input);
    cout << "处理结果: " << output << endl;

    return 0;
}
```

案例 3.4　表达式求值

1. 实验目的

深入理解栈的“后进先出”特性,熟练掌握双栈的定义和使用。

2. 实验内容

利用栈来实现整数的四则运算表达式求值,输入合法的算术表达式(可以包括加、减、乘、除和括号),便可输出相应的计算结果。

3. 实验要求

```
输入:
9+(3-1)×3+10/2
输出:
14
```

4. 实验分析

对于字符串表达式,可以分为左操作数、运算符、右操作数。可以将操作数放入一个栈中,运算符放入另外一个栈中。然后通过运算符的优先级来进行计算。计算时,要先出栈,然后进行计算,然后再将计算的结果入栈进行下次计算。

5. 参考代码

```
#include <iostream>
#include <stack>
#include <string>
#include <cctype>

using namespace std;

//判断运算符优先级
int precedence(char op) {
    if (op == '+' || op == '-') return 1;
    if (op == '*' || op == '/') return 2;
    return 0; //括号的优先级最低
}

//执行运算
int calculate(int a, int b, char op) {
    switch (op) {
        case '+': return a + b;
        case '-': return a - b;
```

```
        case '*': return a * b;
        case '/':
            if (b == 0) {
                throw runtime_error("除数不能为零");
            }
            return a / b;                    //整数除法
        default: return 0;
    }
}

//表达式求值主函数
int evaluateExpression(const string& expr) {
    stack<int> values;                       //存储操作数的栈
    stack<char> ops;                         //存储运算符的栈

    for (int i = 0; i < expr.size(); i++) {
        //跳过空格
        if (expr[i] == ' ') {
            continue;
        }
        //处理数字(可能是多位数)
        else if (isdigit(expr[i])) {
            int num = 0;
            //读取完整的数字
            while (i < expr.size() && isdigit(expr[i])) {
                num = num * 10 + (expr[i] - '0');
                i++;
            }
            values.push(num);
            i--;                             //回退一位,因为循环会自增
        }
        //处理左括号
        else if (expr[i] == '(') {
            ops.push(expr[i]);
        }
        //处理右括号:计算到最近的左括号为止
        else if (expr[i] == ')') {
            while (ops.top() != '(') {
                int b = values.top(); values.pop();
                int a = values.top(); values.pop();
                char op = ops.top(); ops.pop();
                values.push(calculate(a, b, op));
            }
            ops.pop();                       //弹出左括号
        }
        //处理运算符
        else {
            //当前运算符优先级小于或等于栈顶运算符时,先计算栈顶运算符
            while (!ops.empty() && precedence(ops.top()) >= precedence(expr[i])) {
                int b = values.top(); values.pop();
                int a = values.top(); values.pop();
                char op = ops.top(); ops.pop();
                values.push(calculate(a, b, op));
```

```
                }
                ops.push(expr[i]);
            }
        }

        //处理剩余的运算符
        while (!ops.empty()) {
            int b = values.top(); values.pop();
            int a = values.top(); values.pop();
            char op = ops.top(); ops.pop();
            values.push(calculate(a, b, op));
        }

        return values.top();
    }

    int main() {
        string expression;
        cout << "请输入算术表达式: ";
        getline(cin, expression);

        try {
            int result = evaluateExpression(expression);
            cout << "计算结果: " << result << endl;
        } catch (const exception& e) {
            cout << "错误: " << e.what() << endl;
        }

        return 0;
    }
```

3.3 基础实验

实验 3.1 栈的顺序表示和实现

1. 实验内容与要求

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) 初始化顺序栈；

(2) 插入元素；

(3) 删除栈顶元素；

(4) 取栈顶元素；

(5) 遍历顺序栈；

(6) 置空顺序栈。

2. 实验分析

栈的顺序存储结构简称为顺序栈，它是运算受限的顺序表。

对于顺序栈，入栈时，首先判断栈是否为满，栈满的条件为：p->top==MAXNUM-

1,栈满时,不能入栈;否则出现空间溢出,引起错误,这种现象称为上溢。

出栈和读栈顶元素操作,先判断栈是否为空,为空时不能操作,否则产生错误。通常栈空作为一种控制转移的条件。

注意:

(1) 顺序栈中元素用向量存放;

(2) 栈底位置是固定不变的,可设置在向量两端的任意一个端点;

(3) 栈顶位置是随着进栈和出栈操作而变化的,用一个整型量 top(通常称 top 为栈顶指针)来指示当前栈顶位置。

3. 参考代码

```
#include <iostream>
using namespace std;

const int MAXNUM = 20;
typedef int ElemType;

class SqStack {
private:
    ElemType stack[MAXNUM];
    int top;
public:
    SqStack() : top(-1) {}
    void Push(ElemType x) {
        if (top < MAXNUM - 1) {
            stack[++top] = x;
        } else {
            cout << "Overflow!" << endl;
        }
    }
    ElemType Pop() {
        if (top != -1) {
            ElemType x = stack[top--];
            cout << "以前的栈顶数据元素" << x << "已经被删除!" << endl;
            return x;
        } else {
            cout << "Underflow!" << endl;
            return 0;
        }
    }
    ElemType GetTop() {
        if (top != -1) {
            return stack[top];
        } else {
            cout << "Underflow!" << endl;
            return 0;
        }
    }
    void OutStack() {
        cout << endl;
        if (top < 0) {
```

```
            cout << "这是一个空栈!" << endl;
        } else {
            for (int i = top; i >= 0; i--) {
                cout << "第" << i << "个数据元素是: " << stack[i] << endl;
            }
        }
    }
    void setEmpty() {
        top = -1;
    }
};

int main() {
    SqStack * q = nullptr;
    int choice, a;

    do {
        cout << "\n 第一次使用必须初始化! \n\n";
        cout << "主菜单\n";
        cout << "1 初始化顺序栈\n";
        cout << "2 插入一个元素\n";
        cout << "3 删除栈顶元素\n";
        cout << "4 取栈顶元素\n";
        cout << "5 置空顺序栈\n";
        cout << "6 结束程序运行\n";
        cout << "--------------------------------\n";
        cout << "请输入您的选择(1-6): ";
        cin >> choice;
        cout << endl;

        switch (choice) {
            case 1:
                q = new SqStack();
                q->OutStack();
                break;
            case 2:
                cout << "请输入要插入的数据元素: a=";
                cin >> a;
                q->Push(a);
                q->OutStack();
                break;
            case 3:
                q->Pop();
                q->OutStack();
                break;
            case 4:
                cout << "栈顶元素为: " << q->GetTop() << endl;
                q->OutStack();
                break;
            case 5:
                q->setEmpty();
                cout << "顺序栈被置空!" << endl;
                q->OutSta0ck();
```

```
                break;
            case 6:
                delete q;
                exit(0);
        }
    } while (choice <= 6);

    return 0;
}
```

实验 3.2　栈的链式表示和实现

1. 实验内容与要求

编写一个程序实现链栈的各种基本运算，并在此基础上设计一个主程序，完成如下功能：

(1) 初始化链栈；

(2) 链栈置空；

(3) 入栈；

(4) 出栈；

(5) 取栈顶元素；

(6) 遍历链栈。

2. 实验分析

链栈是没有附加头结点的运算受限的单链表。栈顶指针就是链表的头指针。

注意：

(1) LinkStack 结构类型的定义可以方便地在函数体中修改 top 指针本身。

(2) 若要记录栈中元素个数，可将元素个数属性放在 LinkStack 类型中定义。

(3) 链栈中的结点是动态分配的，所以可以不考虑上溢。

3. 参考代码

```
#include <iostream>
using namespace std;

typedef int Elemtype;
typedef struct stacknode {
    Elemtype data;
    struct stacknode * next;
} StackNode;
typedef struct {
    StackNode * top;
} LinkStack;

/* 初始化链栈 */
void InitStack(LinkStack * s) {
    s->top = NULL;
    cout << "链栈初始化成功!" << endl;
}
```

```
/* 链栈置空 */
void setEmpty(LinkStack *s) {
    s->top = NULL;
    cout << "链栈已置空!" << endl;
}

/* 入栈 */
void pushLstack(LinkStack *s, Elemtype x) {
    StackNode *p = new StackNode();
    p->data = x;
    p->next = s->top;
    s->top = p;
}

/* 出栈 */
Elemtype popLstack(LinkStack *s) {
    if (s->top == NULL) {
        cout << "链栈为空,无法出栈!" << endl;
        exit(-1);
    }
    StackNode *p = s->top;
    Elemtype x = p->data;
    s->top = p->next;
    delete p;
    return x;
}

/* 取栈顶元素 */
Elemtype StackTop(LinkStack *s) {
    if (s->top == NULL) {
        cout << "链栈为空!" << endl;
        exit(-1);
    }
    return s->top->data;
}

/* 遍历链栈 */
void Disp(LinkStack *s) {
    cout << "\n链栈中的数据为: " << endl;
    cout << "=======================================\n";
    StackNode *p = s->top;
    while (p != NULL) {
        cout << p->data << endl;
        p = p->next;
    }
    cout << "=======================================\n";
}

int main() {
    cout << "================= 链栈操作===============\n\n";
    LinkStack *s = new LinkStack();
    int choice, n, a;
```

```
    do {
        cout << "\n";
        cout << "第一次使用必须初始化！\n";
        cout << "\n";
        cout << "\n           主菜单          \n";
        cout << "\n     1     初始化链栈      \n";
        cout << "\n     2     入栈            \n";
        cout << "\n     3     出栈            \n";
        cout << "\n     4     取栈顶元素      \n";
        cout << "\n     5     置空链栈        \n";
        cout << "\n     6     结束程序运行    \n";
        cout << "\n--------------------------------\n";
        cout << "请输入您的选择:";
        cin >> choice;
        cout << "\n";

        switch (choice) {
            case 1:
                InitStack(s);
                Disp(s);
                break;
            case 2:
                cout << "输入将要压入链栈的数据的个数: n=";
                cin >> n;
                cout << "依次将" << n << "个数据压入链栈: \n";
                for (int i = 1; i <= n; i++) {
                    cin >> a;
                    pushLstack(s, a);
                }
                Disp(s);
                break;
            case 3:
                cout << "\n 出栈操作开始!\n";
                cout << "输入将要出栈的数据个数: m=";
                cin >> n;
                for (int i = 1; i <= n; i++) {
                  cout << "\n 第" << i << "次出栈的数据是: " <<
                     popLstack(s);
                }
                Disp(s);
                break;
            case 4:
                cout << "\n\n 链栈的栈顶元素为: " << StackTop(s) << endl;
                cout << "\n";
                break;
            case 5:
                setEmpty(s);
                Disp(s);
                break;
            case 6:
                delete s;
                exit(0);
        }
```

```
    } while (choice <= 6);

    return 0;
}
```

实验 3.3　队列的顺序表示和实现

1. 实验内容与要求

编写一个程序实现顺序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；

（2）建立顺序队列；

（3）入队；

（4）出队；

（5）判断队列是否为空；

（6）取队头元素；

（7）遍历队列。

2. 实验分析

队列的顺序存储结构称为顺序队列，顺序队列实际上是运算受限的顺序表。

入队时，将新元素插入 rear 所指的位置，然后将 rear 加 1。出队时，删去 front 所指的元素，然后将 front 加 1 并返回被删元素。

顺序队列中的溢出现象如下。

（1）“下溢”现象。当队列为空时，做出队运算产生的溢出现象。“下溢”是正常现象，常用作程序控制转移的条件。

（2）“真上溢”现象。当队列满时，做进栈运算产生空间溢出的现象。“真上溢”是一种出错状态，应设法避免。

（3）“假上溢”现象。由于入队和出队操作中，头尾指针只增加不减小，致使被删元素的空间永远无法重新利用。当队列中实际的元素个数远远小于向量空间的规模时，也可能由于尾指针已超越向量空间的上界而不能做入队操作。该现象称为“假上溢”现象。

注意：

（1）当头尾指针相等时，队列为空。

（2）在非空队列里，队头指针始终指向队头元素，尾指针始终指向队尾元素的下一位置。

3. 参考代码

```
#include <iostream>
using namespace std;

const int MAXNUM = 100;
typedef int Elemtype;
const int TRUE = 1;
const int FALSE = 0;
```

```
class SqQueue {
private:
    Elemtype queue[MAXNUM];
    int front;
    int rear;
public:
    SqQueue() : front(-1), rear(-1) {}
    bool initQueue() {
        front = -1;
        rear = -1;
        return TRUE;
    }
    bool append(Elemtype x) {
        if (rear >= MAXNUM - 1) return FALSE;
        queue[++rear] = x;
        return TRUE;
    }
    Elemtype Delete() {
        if (front == rear) return 0;
        return queue[++front];
    }
    bool Empty() {
        return (front == rear) ? TRUE : FALSE;
    }
    int gethead() {
        if (front == rear) return 0;
        return queue[front + 1];
    }
    void display() {
        if (front == rear) {
            cout << "队列空!" << endl;
            return;
        }
        cout << "\n 顺序队列依次为: ";
        for (int s = front + 1; s <= rear; s++) {
            cout << queue[s] << "<-";
        }
        cout << "\n 顺序队列的队尾元素所在位置:rear=" << rear << endl;
        cout << "顺序队列的队头元素所在位置:front=" << front << endl;
    }
    void Setsqqueue() {
        int n, m;
        cout << "\n 请输入将要入顺序队列的长度:";
        cin >> n;
        cout << "\n 请依次输入入顺序队列的元素值:\n";
        for (int i = 0; i < n; i++) {
            cin >> m;
            append(m);
        }
    }
};

int main() {
```

```
SqQueue *head = new SqQueue();
int x, y, z, select;

do {
    cout << "\n 第一次使用请初始化！ \n";
    cout << "\n 请选择操作(1--7):\n";
    cout << "====================================\n";
    cout << "1 初始化\n";
    cout << "2 建立顺序队列\n";
    cout << "3 入队\n";
    cout << "4 出队 \n";
    cout << "5 判断队列是否为空\n";
    cout << "6 取队头元素 \n";
    cout << "7 遍历队列\n";
    cout << "====================================\n";
    cin >> select;

    switch (select) {
        case 1:
            head->initQueue();
            cout << "已经初始化顺序队列！ \n";
            break;
        case 2:
            head->Setsqqueue();
            cout << "\n 已经建立队列！ \n";
            head->display();
            break;
        case 3:
            cout << "请输入队的值:\n ";
            cin >> x;
            head->append(x);
            head->display();
            break;
        case 4:
            z = head->Delete();
            cout << "\n 队头元素" << z << "已经出队！ \n";
            head->display();
            break;
        case 5:
            if (head->Empty())
                cout << "队列空\n";
            else
                cout << "队列非空\n";
            break;
        case 6:
            y = head->gethead();
            cout << "队头元素为:" << y << endl;
            break;
        case 7:
            head->display();
            break;
    }
} while (select <= 7);
```

```
    delete head;
    return 0;
}
```

实验 3.4　队列的链式表示和实现

1. 实验内容与要求

编写一个程序实现链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）初始化并建立链队列；

（2）入链队列；

（3）出链队列；

（4）遍历链队列。

2. 实验分析

队列的链式存储结构简称为链队列。它是限制仅在表头删除和表尾插入的单链表。

注意：

（1）和链栈类似，无须考虑判队满的运算及上溢。

（2）在出队算法中，一般只需修改队头指针。但当原队中只有一个结点时，该结点既是队头也是队尾，故删去此结点时亦需修改尾指针，且删去此结点后队列变空。

（3）和单链表类似，为了简化边界条件的处理，在队头结点前可附加一个头结点。

3. 参考代码

```
#include <iostream>
using namespace std;

typedef int ElemType;

class QNode {
public:
    ElemType data;
    QNode * next;

    QNode() : next(nullptr) {}
    QNode(ElemType val) : data(val), next(nullptr) {}
};

class LQueue {
private:
    QNode * front;
    QNode * rear;

public:
    LQueue() : front(nullptr), rear(nullptr) {}

    void create() {
        int n, x;
        cout << "输入将建立链队列元素的个数: n= ";
```

```
        cin >> n;

        QNode *head = new QNode();
        front = head;
        rear = head;

        for (int i = 1; i <= n; ++i) {
            cout << "链队列第 " << i << " 个元素的值为: ";
            cin >> x;
            append(x);
        }
    }

    void append(ElemType x) {
        QNode *s = new QNode(x);
        rear->next = s;
        rear = s;
    }

    ElemType dequeue() {
        if (front == rear) {
            cout << "队列为空!" << endl;
            return 0;
        }

        QNode *p = front->next;
        ElemType x = p->data;

        front->next = p->next;

        if (p->next == nullptr) {
            rear = front;
        }

        delete p;
        return x;
    }

    void display() {
        QNode *p = front->next;
        cout << "\n 链队列元素依次为: ";

        while (p != nullptr) {
            cout << p->data << "-->";
            p = p->next;
        }

        cout << " 遍历链队列结束! \n";
    }
};

int main() {
    LQueue *queue = nullptr;
```

```
    int choice, x;

    do {
        cout << "\n*****第一次操作请选择初始化并建立链队列！*****\n";
        cout << "\n      链队列的基本操作\n";
        cout << "=========================================\n";
        cout << "           主菜单                  \n";
        cout << "=========================================\n";
        cout << "      1    初始化并建立链队列      \n";
        cout << "      2    入链队列                \n";
        cout << "      3    出链队列                \n";
        cout << "      4    遍历链队列              \n";
        cout << "      5    结束程序运行            \n";
        cout << "=========================================\n";

        cin >> choice;

        switch (choice) {
            case 1:
                queue = new LQueue();
                queue->create();
                queue->display();
                break;

            case 2:
                cout << "请输入队列元素的值: x= ";
                cin >> x;
                queue->append(x);
                queue->display();
                break;

            case 3:
                cout << "出链队列元素: x=" << queue->dequeue() << endl;
                queue->display();
                break;

            case 4:
                queue->display();
                break;

            case 5:
                delete queue;
                return 0;
        }
    } while (choice <= 5);

    return 0;
}
```