广州大学学生实验报告

开课实验室： 电子楼 416B　　2018 年 4 月 29 日

学院	计算机科学与网络工程学院	年级、专业、班	软件 171	姓名	谢金宏	学号	1706300001
实验课程名称	数据结构					成绩	
实验项目名称	实验一　链式存储结构的基本操作					指导老师	杜娇

一、实验目的

掌握单链表、链式堆栈和链式队列的定义及基本操作。

二、使用仪器、器材

1. 使用 Windows 操作系统的微型计算机一台。
2. 符合 C++11 标准的 C++编程软件。

三、实验内容及原理

（一）单链表的定义及基本操作

1. 用带表头的链表存放输入的数据，每读入一个数，按升序顺序插入到链表中，链表中允许两个结点有相同值。链表的头结点存放链表后面的结点个数，初始化时就生成头结点（初值为 0）。
2. 在上述带表头的链表中删除第 i 个结点或删除数值为 item 的结点。
3. 链表翻转是把数据逆序（变成降序），注意，头结点不动。翻转后要再翻转一次，恢复升序后才能插入新元素，否则会出错。
4. 设 A 与 B 分别为两个带有头结点的有序循环链表（所谓有序是指链接点按数据域值大小链接，本题不妨设按数据域值从小到大排列），list1 和 list2 分别为指向两个链表的指针。请写出并在计算机上实现将这两个链表合并为一个带头结点的有序循环链表的算法。

（二）链式堆栈的定义及基本操作

5. 先定义堆栈的几个基本操作，再设计一主函数利用堆栈的操作完成以下功能：假设一个算术表达式中可以包含三种括号：()[]{}，且这三种括号可以按任意次序嵌套使用（如：...[...{...}...[...]...]...(...)）。编写判别给定表达式中所含括号是否正确配对出现的算法，已知表达式已存入数据元素为字符的单链表中。

（三）链式队列的定义及基本操作

6. 先定义队列的几个基本操作，再设计一主函数利用队列的操作完成以下功能：键盘输入的字符可以临时存入键盘的缓冲区中。为了充分利用缓冲区的空间，往往将缓冲区设计成链式循环队列的结构，并为循环队列结构的缓冲区设置一个队首指针和一个队尾指针。每输入一个字符到缓冲区中，就将尾指针后移，链入缓冲区的循环队列之中；每输出一个字符号，就将队头指针前移，将它从缓冲队列中删除。假设有两个进程同时存在于一个应用程序中，第一个进程连续在屏幕上显示字符“X”，第二个进程不断检查键盘上是否有输入，若有则读入用户键入的字符，将其保存到键盘缓冲区中。

四、实验过程原始数据记录

实验使用的源代码如下：


LinkList.cpp

CycleLinkList.cpp

Stack.cpp

Queue.cpp

（一）单链表的定义及基本操作

第 1、2、3 小项使用以下单链表代码。

基本代码

```
// 结点
struct Node {
    int elem;
    Node * next;
};


// 单链表
// 链表中头节点保存数据结点的个数，数据结点从 1 开始编号。
typedef Node *LinkList;

// 构造链表
void InitList(LinkList & list)
{
    list = new Node;
    list->elem = 0;
    list->next = nullptr;
}
```

```cpp
// 析构链表
void DestroyList(LinkList & list)
{
    Node * cur = list, * next;
    while (cur) {
        next = cur->next;
        delete cur;
        cur = next;
    }
    list = nullptr;
}

// 向链表中升序插入数据
void ListInsert(LinkList & list, int elem)
{
    Node * cur = list;
    while (cur->next) {
        if (elem <= cur->next->elem) break;
        cur = cur->next;
    }
    Node * neo = new Node;
    neo->elem = elem;
    neo->next = cur->next;
    cur->next = neo;
    list->elem += 1;
}
```

```cpp
// 删除链表中序号为 pos 的结点
void ListRemoveAt(LinkList & list, int pos)
{
    if (pos < 0 || pos > list->elem) return;  // 边界检查

    Node * pre = list;
    while (--pos) pre = pre->next;  // 将 pre 指针定位到待删除结点的前一个结
点

    Node * tbd = pre->next;  // 将 tbd 指针定位到待删除结点
    pre->next = tbd->next;
    delete tbd;

    list->elem -= 1;
}

// 删除链表中值为 val 的结点
void ListRemoveVal(LinkList & list, int val)
{
    Node * pre = list, * cur = list->next;
    while (cur) {
        Node * nxt = cur->next;
        if (cur->elem == val) {
            delete cur; list->elem -= 1;
            pre->next = nxt, cur = pre;
        }
        pre = cur, cur = nxt;
    }
}

// 反转链表
void ListReverse(LinkList & list)
{
    Node * pre = nullptr, * cur = list->next, * nxt = nullptr;
    while (cur) {
        nxt = cur->next;
        cur->next = pre;
        pre = cur, cur = nxt;
    }  // 离开循环时，pre 指针指向反转前链表的末尾结点，即新链表的首元结点
    list->next = pre;
}
```

```
// 输出链表
void ListDump(LinkList & list)
{
    cout << "[" << list->elem << "] ";  // 输出头节点的存放的值，即链表中元
素的个数

    Node * cur = list->next;
    while (cur) {
        cout << cur->elem;
        cout << (cur->next ? ' ' : '\n');
        cur = cur->next;
    } // 输出数据结点中存放的值
}
```

测试用例

```
// 测试用例
void test()
{
    LinkList a; InitList(a);

    cout << "初始化链表：" << endl;
    ListInsert(a, 5), ListInsert(a, 4), ListInsert(a, 6);
    ListInsert(a, 3), ListInsert(a, 10), ListInsert(a, -2);
    ListDump(a);

    cout << "测试按序号删除结点功能，删除序号 1、3、5 的结点。" << endl;
    ListRemoveAt(a, 1), ListRemoveAt(a, 3), ListRemoveAt(a, 5);
    ListDump(a);

    cout << "测试按值删除结点功能，插入 2 个值为 12 的结点和 1 个值为 6 的结点，删
除值为 4 或 12 的结点。" << endl;
    ListInsert(a, 12); ListInsert(a, 12); ListInsert(a, 6);
    ListDump(a);
    ListRemoveVal(a, 4); ListRemoveVal(a, 12);
    ListDump(a);

    cout << "测试翻转链表功能。" << endl;
    ListReverse(a);
    ListDump(a);
    ListReverse(a);
    ListDump(a);

    cout << "综合测试。" << endl;
    DestroyList(a), InitList(a);
    ListInsert(a, -8), ListInsert(a, 12), ListInsert(a, 6);
    ListReverse(a);
    ListDump(a);

    DestroyList(a);
}
```

命令行交互

```cpp
int main()
{
    cout << "单链表定义及基本操作验证性实验" << endl << endl;
    cout << "链表的数据类型定义为 INT，第一个数据结点的下标为 1。" << endl;
    cout << "请使用 Ctrl+Z[ENTER]标志输入结束。" << endl << endl;

    LinkList list; InitList(list);
    cout << "请输入链表的初始值：" << endl;
    {
        int neo; while (cin >> neo) {
            ListInsert(list, neo);
            ListDump(list);
        }
        cin.clear();
    }

    cout << "请输入欲删除结点的序号：" << endl;
    {
        int pos; while (cin >> pos) {
            ListRemoveAt(list, pos);
            ListDump(list);
        }
        cin.clear();
    }

    cout << "请输入欲删除结点的值：" << endl;
    {
        int val; while (cin >> val) {
            ListRemoveVal(list, val);
            ListDump(list);
        }
        cin.clear();
    }

    cout << "翻转链表" << endl;
    ListReverse(list), ListDump(list);
    cout << "再次反转链表" << endl;
    ListReverse(list); ListDump(list);

    cout << "程序结束" << endl;
}
```

第 4 小项使用以下循环链表代码。

基本代码

```cpp
// 循环链表
// 链表中头节点保存数据结点的个数，数据结点从 1 开始编号。
typedef Node *CycleLinkList;

// 构造链表
void InitList(CycleLinkList & list)
{
    list = new Node;
    list->elem = 0;
    list->next = list;
}


// 析构链表
void DestroyList(CycleLinkList & list)
{
    Node * cur = list->next, * next;
    while (cur != list) {
        next = cur->next;
        delete cur;
        cur = next;
    }  // 删除数据结点
    delete list, list = nullptr;  // 删除头节点
}

// 向链表中升序插入数据
void ListInsert(CycleLinkList & list, int elem)
{
    Node * cur = list;
    while (cur->next != list) {
        if (elem <= cur->next->elem) break;
        cur = cur->next;
    }
    Node * neo = new Node;
    neo->elem = elem;
    neo->next = cur->next;
    cur->next = neo;
    list->elem += 1;
}
```

```cpp
// 删除链表中序号为 pos 的结点
void ListRemoveAt(CycleLinkList & list, int pos)
{
    if (pos < 0 || pos > list->elem) return;  // 边界检查

    Node * pre = list;
    while (--pos) pre = pre->next;  // 将 pre 指针定位到待删除结点的前一个结点

    Node * tbd = pre->next;  // 将 tbd 指针定位到待删除结点
    pre->next = tbd->next;
    delete tbd;

    list->elem -= 1;
}

// 删除链表中值为 val 的结点
void ListRemoveVal(CycleLinkList & list, int val)
{
    Node * pre = list, * cur = list->next;
    while (cur != list) {
        Node * nxt = cur->next;
        if (cur->elem == val) {
            delete cur; list->elem -= 1;
            pre->next = nxt, cur = pre;
        }
        pre = cur, cur = nxt;
    }
}

// 反转链表
void ListReverse(CycleLinkList & list)
{
    Node * pre = list, * cur = list->next, * nxt = nullptr;
    while (cur != list) {
        nxt = cur->next;
        cur->next = pre;
        pre = cur, cur = nxt;
    }  // 离开循环时，pre 指针指向反转前链表的末尾结点，即新链表的首元结点
    list->next = pre;
}
```

```cpp
// 有序合并循环链表
CycleLinkList ListMerge(CycleLinkList list_a, CycleLinkList list_b)
{
    CycleLinkList list_c = new Node;  // 新链表的头节点

    list_c->elem = list_a->elem + list_b->elem;  // 合并链表后的结点个数为合并前两链表结点个数之和
    Node * cur_a = list_a->next, * cur_b = list_b->next;  // cur 指向旧链表中表头结点的下一个结点
    Node * cur_c = list_c;  // 在后续循环中将在 cur_c 后插入新节点
    while (cur_a != list_a && cur_b != list_b)
    {
        Node *& less = cur_a->elem < cur_b->elem ? cur_a : cur_b;  // 选择两个结点中较小的一个
        Node * neo = new Node{less->elem, nullptr};  // 将较小结点的值复制到新节点中
        cur_c->next = neo, cur_c = neo;  // 将新节点插入新链表中
        less = less->next;
    }
    while (cur_a != list_a) {
        Node * neo = new Node{cur_a->elem, nullptr};
        cur_c->next = neo, cur_c = neo;
        cur_a = cur_a->next;
    }
    while (cur_b != list_b) {
        Node * neo = new Node{cur_b->elem, nullptr};
        cur_c->next = neo, cur_c = neo;
        cur_b = cur_b->next;
    }
    cur_c->next = list_c;
    return list_c;
}
```

```
// 输出链表
void ListDump(CycleLinkList & list)
{
    cout << "[" << list->elem << "] ";  // 输出头节点的存放的值，即链表中元
素的个数

    Node * cur = list->next;
    while (cur != list) {
        cout << cur->elem;
        cout << (cur->next != list ? ' ' : '\n');
        cur = cur->next;
    }  // 输出数据结点中存放的值
}
```

用户交互工具

```
// 向用户输出 hint 提示字符串，提示用户输入链表数据
void ListInput(CycleLinkList list, string hint)
{
    cout << hint << endl;
    int neo; while (cin >> neo) {
        ListInsert(list, neo);
    }
    ListDump(list);  // 在结束输入时输出链表的值
    cin.clear();
}
```

测试用例

```
// 测试用例
void test()
{
    CycleLinkList a; InitList(a);

    cout << "初始化链表：" << endl;
    ListInsert(a, 5), ListInsert(a, 4), ListInsert(a, 6);
    ListInsert(a, 3), ListInsert(a, 10), ListInsert(a, -2);
    ListDump(a);

    cout << "测试按序号删除结点功能，删除序号 1、3、5 的结点。" << endl;
    ListRemoveAt(a, 1), ListRemoveAt(a, 3), ListRemoveAt(a, 5);
    ListDump(a);

    cout << "测试按值删除结点功能，插入 2 个值为 12 的结点和 1 个值为 6 的结点，删
除值为 4 或 12 的结点。" << endl;
    ListInsert(a, 12); ListInsert(a, 12); ListInsert(a, 6);
    ListDump(a);
    ListRemoveVal(a, 4); ListRemoveVal(a, 12);
    ListDump(a);

    cout << "测试翻转链表功能。" << endl;
    ListReverse(a);
    ListDump(a);
    ListReverse(a);
    ListDump(a);

    cout << "综合测试。" << endl;
    DestroyList(a), InitList(a);
    ListInsert(a, -8), ListInsert(a, 12), ListInsert(a, 6);
    ListReverse(a);
    ListDump(a);

    DestroyList(a);
}
```

命令行交互

```
int main()
{
    cout << "循环链表定义及基本操作验证性实验" << endl << endl;
    cout << "链表的数据类型定义为 INT，第一个数据结点的下标为 1。" << endl;
    cout << "请使用 Ctrl+Z[ENTER]标志输入结束。" << endl << endl;

    CycleLinkList list_a; InitList(list_a);
    CycleLinkList list_b; InitList(list_b);

    ListInput(list_a, "请输入链表 A 的值：");
    ListInput(list_b, "请输入链表 B 的值：");

    CycleLinkList list_c = ListMerge(list_a, list_b);
    cout << "将链表 A、B 合并为链表 C：" << endl;
    ListDump(list_c);

    cout << "程序结束" << endl;
}
```

（二）链式堆栈的定义及基本操作

基本代码

```
// 结点
struct Node
{
    char elem;
    Node * next = nullptr;
};

// 单链表
// 链表中头节点保存数据结点的个数，数据结点从 1 开始编号。
typedef Node *LinkList;

// 栈
struct Stack
{
    Node * base, * top;  // 指向栈底与栈顶元素的指针
};
```

```
// 构造链表
void InitList(LinkList & list)
{
    list = new Node;
    list->elem = 0;
    list->next = nullptr;
}


// 析构链表
void DestroyList(LinkList & list)
{
    Node * cur = list, * next;
    while (cur) {
        next = cur->next;
        delete cur;
        cur = next;
    }
    list = nullptr;
}

// 在链表末尾插入值为 val 的结点
void ListAppend(LinkList & list, char val)
{
    Node * tail = list;
    while (tail->next) {
        tail = tail->next;
    }
    list->elem += 1;
    tail->next = new Node{val, nullptr};
}

// 构造栈
void InitStack(Stack & stk)
{
    stk.base = stk.top = nullptr;
}
```

```cpp
// 判断栈是否为空
bool StackEmpty(Stack & stk)
{
    return stk.base == nullptr;
}

// 向栈中压入值为 val 的结点
Node* StackPush(Stack & stk, char val)
{
    Node * neo = new Node {val, nullptr};
    if (stk.top == nullptr) {
        stk.base = stk.top = neo;
    }
    else {
        stk.top->next = neo;
        stk.top = neo;
    }
}

// 取栈顶结点的值
char StackTop(Stack & stk)
{
    return stk.top->elem;
}

// 弹出栈顶元素
void StackPop(Stack & stk)
{
    if (stk.base == stk.top) {  // 正在弹出栈中最后一个结点
        delete stk.top;
        stk.base = stk.top = nullptr;  // 最后一个结点被弹出
    }
    else {
        Node * pre = stk.base, * tail = stk.base->next;
        while (tail->next != nullptr) {
            pre = pre->next, tail = tail->next;
        }  // 定位栈顶指针的前一位指针及栈顶指针
        delete tail; pre->next = nullptr;  // 弹出栈顶
        stk.top = pre;  // 设置新的栈顶指针
    }
}
```

```cpp
// 析构栈
void DestroyStack(Stack & stk)
{
    while (!StackEmpty(stk))
        StackPop(stk);
}
```

测试用例

```cpp
// 测试用例
// 将一串字符压入栈后弹出，测试栈是否能正常工作
void test()
{
    Stack stk; InitStack(stk);
    char ch; while (cin >> ch)
    {
        StackPush(stk, ch);
    }
    while (!StackEmpty(stk)) {
        cout << StackTop(stk) << ' ';
        StackPop(stk);
    }
    cout << endl;
    DestroyStack(stk);
}
```

进行括号匹配的算法

```cpp
    LinkList list; InitList(list);  // 初始化用于存放值的单链表
    {
        cout << "请输入等待匹配的括号表达式: " << endl;
        cout << "（按 Ctrl+Z[ENTER]结束输入）" << endl;
        char ch; while (cin.get(ch) && !cin.eof()) {
            if (!isspace(ch)) {  // 若当前输入的字符不是“ ”或“\n”等空格字符
                ListAppend(list, ch);  // 将字符输入链表中
            }
        }
    }
```

```
    Stack stack; InitStack(stack);  // 初始用于解决括号匹配问题的栈

    bool match = true;  // 假设括号能成功匹配
    Node * cur = list->next;  // 等待匹配的括号结点

    while (match && cur)
    {
        char ch = cur->elem;  // 等待匹配的括号字符
        if (ch == '(' || ch == '[' || ch == '{') {
            StackPush(stack, ch);  // 左括号入栈
        }
        else {  // 处理右括号
            if (StackEmpty(stack)) {  // 栈为空则匹配失败
                match = false; continue;
            }
            char top = StackTop(stack);
            if ((ch == ')' && top == '(') || (ch == ']' && top == '[') || (ch
== '}' && top == '{') ) {
                StackPop(stack);  // 弹出栈顶结点
            } else {
                match = false;
            }
        }
        cur = cur->next;  // 前进到下一个结点
    }

    if (!StackEmpty(stack)) match = false;

    cout << (match ? "匹配成功" : "匹配失败") << endl;

    DestroyList(list), DestroyStack(stack);
```

（三）链式队列的定义及基本操作

基本代码

```
// 链式循环队列
// 首元结点用于保存队列长度，队头结点是首元结点的后一个结点
// 末尾结点的下一个结点是首元结点
typedef Node *Queue;
```

```
// 构造队列
void InitQueue(Queue & que)
{
    que = new Node;
    que->elem = 0;
    que->next = que;  // 初始化为空队列
}

// 判断队列是否为空
bool QueueEmpty(Queue & que)
{
    return que->elem == 0;
}

// 获取队头结点的值
char QueueFront(Queue & que)
{
    return que->next->elem;
}

// 获取队列长度
int QueueLength(Queue & que)
{
    return que->elem;
}

// 向队尾添加值为 val 的结点
void QueuePush(Queue & que, char val)
{
    Node * rear = que;
    while (rear->next != que) {
        rear = rear->next;
    }
    rear->next = new Node{val, que};  // 向队尾添加结点
    que->elem += 1;
}
```

```
// 队头结点出队
void QueuePop(Queue & que)
{
    if (que->next != que) {  // 若队列不为空
        Node * head = que->next;
        que->next = head->next;
        delete head;
        que->elem -= 1;
    }
}

// 析构队列
void DestroyQueue(Queue & que)
{
    while (!QueueEmpty(que)) {
        QueuePop(que);
    }
    delete que; que = nullptr;
}

// 输出队列中的结点
void QueueDump(Queue & que)
{
    if (que->next == que) {
        cout << "队列为空" << endl;
        return;
    }

    cout << que << ' ' << que->next << endl;

    cout << que->next->elem;  // 输出队头结点的值
    for (Node * cur = que->next->next; cur != que; cur = cur->next) {
        cout << ' ' << cur->elem;  // 输出队头结点其后结点的值
    }
    cout << endl;
}
```

测试用例

```
// 测试用例
void test()
{
    Queue que; InitQueue(que);
    QueueDump(que);

    QueuePush(que, '1'), QueuePush(que, '2'), QueuePush(que, '3'),
QueuePush(que, '4'),
    QueuePush(que, '5'), QueuePush(que, '6'), QueuePush(que, '7'),
QueuePush(que, '8');

    while (!QueueEmpty(que)) {
        QueuePop(que);
        QueueDump(que);
    }

    QueuePush(que, 'x');
    DestroyQueue(que);
}
```

“模拟键盘缓冲区”

```cpp
int main()
{
    cout << "链式队列定义及基本操作实验" << endl;
    cout << "键盘缓冲区模拟：每读入 10 个字符向屏幕输出一行字符。" << endl;
    cout << "为了与用户输入区分，缓冲区的输出前附加有“>> ”记号。" << endl;
    cout << "按 Ctrl+Z[ENTER]结束输入。" << endl;

    Queue que; InitQueue(que);  // 初始化用于模拟键盘缓冲区的链式队列
    char ch; while (cin.get(ch) && !cin.eof()) {
        QueuePush(que, ch);
        if (QueueLength(que) == 10) {  // 每读入 10 个字符输出一次
            cout << ">> ";
            while (!QueueEmpty(que)) {
                cout << QueueFront(que);
                QueuePop(que);
            }
            cout << endl;
        }
    }

    if (!QueueEmpty(que)) {  // 如果结束输入时队列中还有数据
        cout << ">> ";
        while (!QueueEmpty(que)) {
            cout << QueueFront(que);
            QueuePop(que);
        }  // 将队列中的数据输出
    }
}
```

五、实验结果及分析

本次实验整体完成度较好。

（一）单链表的定义及基本操作

程序完成了题目的要求，并通过了自己定义的测试用例。

（二）链式堆栈顶定义及基本操作

程序完成了题目的要求，并通过了自己定义的测试用例。

（三）链式队列的定义及基本操作

由于不熟悉 Windows 上的多线程编程和 TUI 编写，没有实现题目要求中的“多线程”。但实现了链式队列，部分完成了题目要求，通过了自己定义的测试用例。