南開大學
Nankai University

数据结构实验报告 2

链表

姓　　名：南亚宏

学　　号：2311788

目录

2025 年 10 月 25 日

一、 题目一

1.1 题目表述

令 a 和 b 的类型为 extendedChain。
1)编写一个非成员方法 meld,它生成一个新的扩展的链表 c,它从 a 的首元素开始,交替地包含 a 和 b 的元素。如果一个链表的元素取完了,就把另一个链表的剩余元素附加到新的扩展链表 c 中。方法的复杂度应与链表 a 和 b 的长度具有线性关系。
2)证明方法具有线性复杂度。
3)使用自己的测试数据检验方法的正确性。

1.2 代码 1 的测试用例:

a = [], b = [] → c = []

a = [], b = [1,2,3] → c = [1,2,3]

a = [7,8], b = [] → c = [7,8]

a = [1,3,5], b = [2,4,6] → c = [1,2,3,4,5,6]

a = [10,20,30,40], b = [11] → c = [10,11,20,30,40]

a = [-3,0,5], b = [-2,-1,6,7] → c = [-3,-2,0,-1,5,6,7]

用表格列出：输入 a、输入 b、程序输出 c、期望输出、是否正确。说明你的实现是 **新建节点** 还是 **复用节点**。给出复杂度分析（说明为什么是 O(n+m)）。

1.3 思路:

首先定义一个链表节点结构和链表类，为链表的基本操作如添加元素和获取头节点提供支持。然后创建一个函数来合并两个链表。这个函数交替从两个输入链表中取出元素，并添加到新的链表中。如果一个链表的元素先被取完，就将另一个链表的剩余元素直接追加到新链表的末尾。

1.4 代码:

```
#include <iostream>
```

```cpp
#include <string>
using namespace std;

// 定义链表节点结构
struct Node {
    int data;
    Node* next;
    Node(int val) : data(val), next(nullptr) {}
};

// 定义扩展链表类
class ExtendedChain {
private:
    Node* head;
    Node* tail;

public:
    ExtendedChain() : head(nullptr), tail(nullptr) {}

    // 添加元素到链表尾部
    void append(int val) {
        Node* newNode = new Node(val);
        if (tail == nullptr) {
            head = tail = newNode;
        } else {
            tail->next = newNode;
            tail = newNode;
        }
    }

    // 获取链表头节点
    Node* getHead() const {
        return head;
    }

    // 打印链表
    void print() const {
        if (head == nullptr) {
            cout << "[]" << endl;
            return;
        }
```

```
        cout << "[";
        Node* current = head;
        while (current) {
            cout << current->data;
            if (current->next) {
                cout << ", ";
            }
            current = current->next;
        }
        cout << "]" << endl;
    }
};

// meld
ExtendedChain meld(const ExtendedChain& a, const
ExtendedChain& b) {
    ExtendedChain c;
    Node* aNode = a.getHead();
    Node* bNode = b.getHead();

    while (aNode && bNode) {
        c.append(aNode->data);
        aNode = aNode->next;
        c.append(bNode->data);
        bNode = bNode->next;
    }

    // 添加剩余的节点
    while (aNode) {
        c.append(aNode->data);
        aNode = aNode->next;
    }
    while (bNode) {
        c.append(bNode->data);
        bNode = bNode->next;
    }

    return c;
}

// 测试
void testMeld() {
```

```
ExtendedChain a, b, c;

// 测试用例 1
a = ExtendedChain();
b = ExtendedChain();
c = meld(a, b);
cout << "a = [], b = [] -> c = ";
c.print();

// 测试用例 2
a = ExtendedChain();
b = ExtendedChain();
b.append(1); b.append(2); b.append(3);
c = meld(a, b);
cout << "a = [], b = [1,2,3] -> c = ";
c.print();

// 测试用例 3
a = ExtendedChain();
a.append(7); a.append(8);
b = ExtendedChain();
c = meld(a, b);
cout << "a = [7,8], b = [] -> c = ";
c.print();

// 测试用例 4
a = ExtendedChain();
a.append(1); a.append(3); a.append(5);
b = ExtendedChain();
b.append(2); b.append(4); b.append(6);
c = meld(a, b);
cout << "a = [1,3,5], b = [2,4,6] -> c = ";
c.print();

// 测试用例 5
a = ExtendedChain();
a.append(10); a.append(20); a.append(30); a.append(40);
b = ExtendedChain();
b.append(11);
c = meld(a, b);
cout << "a = [10,20,30,40], b = [11] -> c = ";
c.print();
```

```
    // 测试用例 6
    a = ExtendedChain();
    a.append(-3); a.append(0); a.append(5);
    b = ExtendedChain();
    b.append(-2); b.append(-1); b.append(6); b.append(7);
    c = meld(a, b);
    cout << "a = [-3,0,5], b = [-2,-1,6,7] -> c = ";
    c.print();
}

int main() {
    testMeld();
    return 0;
}
```

实现说明
在 meld 方法中，创建新的节点来构建新的链表 c，而不是复用 a 和 b 的节点。这样可以保证原始链表 a 和 b 不被修改。

复杂度分析
时间复杂度：meld 方法的时间复杂度为 O(n+m)，其中 n 和 m 分别是链表 a 和 b 的长度。因为只需要遍历两个链表一次，将它们的元素交替添加到新的链表 c 中。

空间复杂度：meld 方法的空间复杂度为 O(n+m)。因为需要创建一个新的链表 c 来存储合并后的结果。

1.5 测试用例截图

```
PS C:\Users\Macro\Desktop\hw2> cd .\hw2.1
PS C:\Users\Macro\Desktop\hw2\hw2.1> ./main.exe
  a = [], b = [] -> c = []
  a = [], b = [1,2,3] -> c = [1, 2, 3]
  a = [7,8], b = [] -> c = [7, 8]
  a = [1,3,5], b = [2,4,6] -> c = [1, 2, 3, 4, 5, 6]
  a = [10,20,30,40], b = [11] -> c = [10, 11, 20, 30, 40]
  a = [-3,0,5], b = [-2,-1,6,7] -> c = [-3, -2, 0, -1, 5, 6, 7]
PS C:\Users\Macro\Desktop\hw2\hw2.1>
```

输入 a	输入 b	输出 c	期望输出	是否正确
空	空	空	空	√
空	1,2,3	1,2,3	1,2,3	√
7,8	空	7,8	7,8	√
1,3,5	2,4,6	1,2,3,4,5,6	1,2,3,4,5,6	√
10，20，30，40	11	10,11,20,30,40	10,11,20,30,40	√
-3,0,5	-2,-1,6,7	-3,-2,0,-1,5,6,7	-3,-2,0,-1,5,6,7	√

二、 题目二

2.1 题目表述

使用带有头节点的双向循环链表解决问题：
1.令 c 的类型为扩展链表 extendedChain。
1)编写一个非成员方法 split(a,b),它生成两个扩展链表 a 和 b。a 包含 c 中索引为奇数的元素，b 包含 c 中其余的元素。这个方法不能改变 c。
2)计算方法的复杂度。
3)使用测试数据检验方法的正确性。
2.编写方法 chain<T> :: split,它与上面的函数类似。然而,它用输入链表*this 的空间建立了链表 a 和 b。
用表格列出：输入 a、输入 b、程序输出 c、期望输出、是否正确。说明你的实现是**不改原表**还是**摘链**。给出复杂度分析。

2.2 代码 2 的测试用例:

c = [] → a = []，b = []

c = [7] → a = [7]，b = []

c = [1,2] → a = [1]，b = [2]

c = [1,2,3,4,5] → a = [1,3,5]，b = [2,4]

c = [10,20,30,40] → a = [10,30]，b = [20,40]

c = [-1,-1,0,7] → a = [-1,0]，b = [-1,7]

2.3 思路:

第一问

从链表 c 的头节点开始，遍历整个链表，获取每个节点的数据；在遍历过程中，根据节点索引的奇偶性（从 0 开始计数），决定将当前节点的数据插入到新链表 a 还是 b，插入到相应的新链表尾部。遍历结束后，两个新链表 a 和 b 分别包含了原链表中索引为奇数和偶数的元素，而原链表 c 保持不变。

第二问

方法 split 遍历当前链表 this，根据索引的奇偶性将节点重新分配到 a 和 b 中。再使用 lastA 和 lastB 分别跟踪 a 和 b 链表的最后一个节点，以便正确地连接新节点。遍历完成后，清空原链表 this，使其头节点的 next 和 prev 指向自身。

2.4 代码:

第一问

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

// 定义双向循环链表节点
struct Node {
    int data;
    Node* prev;
    Node* next;

    Node(int val) : data(val), prev(nullptr), next(nullptr) {}
};

// 扩展链表类
class ExtendedChain {
private:
    Node* head; // 头节点

public:
    ExtendedChain() {
        head = new Node(0); // 创建头节点
        head->prev = head;
        head->next = head;
    }
```

```
    // 析构函数，释放链表内存
    ~ExtendedChain() {
        if (head) {
            Node* current = head->next;
            while (current != head) {
                Node* temp = current;
                current = current->next;
                delete temp;
            }
            delete head;
        }
    }

    // 插入元素到链表尾部
    void insert(int val) {
        Node* newNode = new Node(val);
        newNode->prev = head->prev;
        newNode->next = head;
        head->prev->next = newNode;
        head->prev = newNode;
    }

    // 打印链表
    void print() const {
        Node* current = head->next;
        while (current != head) {
            cout << current->data << " ";
            current = current->next;
        }
    }

    // 获取头节点的指针
    Node* getHead() const {
        return head;
    }
};

// 非成员方法 split
void split(const ExtendedChain& c, ExtendedChain& a,
ExtendedChain& b) {
    int index = 0;
    Node* current = c.getHead()->next; // 使用公共方法获取头节点
```

```
    while (current != c.getHead()) {
        if (index % 2 == 0) {
            a.insert(current->data);
        } else {
            b.insert(current->data);
        }
        current = current->next;
        index++;
    }
}

// 测试代码
int main() {
    // 测试用例
    vector<vector<int>> testCases = {
        {}, {7}, {1, 2}, {1, 2, 3, 4, 5}, {10, 20, 30, 40}, {-
1, -1, 0, 7}
    };

    for (const auto& testCase : testCases) {
        ExtendedChain c;
        for (int val : testCase) {
            c.insert(val);
        }

        ExtendedChain a, b;
        split(c, a, b);

        cout << "c = [";
        c.print();
        cout << "] → a = [";
        a.print();
        cout << "], b = [";
        b.print();
        cout << "]" << endl;
    }

    return 0;
}
```

实现说明：

不改变原表。通过遍历原链表并根据条件将元素插入到两个新链表中，原链表的结构和内容保持不变。

时间复杂度：split 方法的时间复杂度为 O(n)，其中 n 是链表 c 的长度。由于我们遍历了整个链表一次，以确定每个元素应该插入到哪个新链表中。

插入操作的时间复杂度为 O(1)，因为每次插入都是直接在链表尾部进行的，不需要移动其他元素。

整体时间复杂度就是 O(n)。

空间复杂度：O(n)，因为创建了两个新的链表来存储结果，这两个链表总共包含与原链表相同数量的节点。此外没有使用额外的数据结构来存储中间结果，因此空间复杂度主要来自于新创建的链表。

原表 c	输出 a	输出 b	期望 a	期望 b	是否正确
空	空	空	空	空	√
7	7	空	7	空	√
1,2	1	2	1	2	√
1,2,3,4,5	1,3,5	2,4	1,3,5	2,4	√
10,20,30,40	10,30	20,40	10,30	20,40	√
-1,-1,0,7	-1,0	-1,7	-1,0	-1,7	√

第二问

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

// 定义双向循环链表节点
struct Node {
    int data;
    Node* prev;
    Node* next;

    Node(int val) : data(val), prev(nullptr), next(nullptr) {}
};

// 扩展链表类
class ExtendedChain {
private:
    Node* head; // 头节点

public:
    ExtendedChain() {
        head = new Node(0); // 创建头节点
        head->prev = head;
```

```
    head->next = head;
}

// 析构函数，释放链表内存
~ExtendedChain() {
    if (head) {
        Node* current = head->next;
        while (current != head) {
            Node* temp = current;
            current = current->next;
            delete temp;
        }
        delete head;
    }
}

// 插入元素到链表尾部
void insert(int val) {
    Node* newNode = new Node(val);
    newNode->prev = head->prev;
    newNode->next = head;
    head->prev->next = newNode;
    head->prev = newNode;
}

// 打印链表
void print() const {
    Node* current = head->next;
    while (current != head) {
        cout << current->data << " ";
        current = current->next;
    }
}

// 获取头节点的指针
Node* getHead() const {
    return head;
}

// 成员方法 split
void split(ExtendedChain& a, ExtendedChain& b) {
    Node* current = head->next; // 当前节点
```

```
Node* lastA = nullptr;      // a 链表的最后一个节点
Node* lastB = nullptr;      // b 链表的最后一个节点
int index = 0;

while (current != head) {
    Node* next = current->next; // 保存下一个节点

    if (index % 2 == 0) {
        // 将当前节点加入到 a 链表
        if (lastA == nullptr) {
            a.head->next = current;
            a.head->prev = current;
            current->next = a.head;
            current->prev = a.head;
        } else {
            lastA->next = current;
            current->prev = lastA;
            current->next = a.head;
            a.head->prev = current;
        }
        lastA = current;
    } else {
        // 将当前节点加入到 b 链表
        if (lastB == nullptr) {
            b.head->next = current;
            b.head->prev = current;
            current->next = b.head;
            current->prev = b.head;
        } else {
            lastB->next = current;
            current->prev = lastB;
            current->next = b.head;
            b.head->prev = current;
        }
        lastB = current;
    }

    current = next; // 移动到下一个节点
    index++;
}

// 清空原链表
```

```
        head->next = head;
        head->prev = head;
    }
};

// 测试代码
int main() {
    // 测试用例
    vector<vector<int>> testCases = {
        {}, {7}, {1, 2}, {1, 2, 3, 4, 5}, {10, 20, 30, 40}, {-
1, -1, 0, 7}
    };

    for (const auto& testCase : testCases) {
        ExtendedChain c;
        for (int val : testCase) {
            c.insert(val);
        }

        ExtendedChain a, b;
        c.split(a, b);

        cout << "c = [";
        c.print();
        cout << "] → a = [";
        a.print();
        cout << "], b = [";
        b.print();
        cout << "]" << endl;
    }

    return 0;
}
```

实现说明：
摘链。即通过重新连接节点来形成新的链表 a 和 b，而不是创建新的节点。
复杂度分析：
时间复杂度 O(n)，其中 n 是链表的长度。因为我们遍历了整个链表一次。
空间复杂度 O(1)，因为我们没有使用额外的空间来存储节点，只是重新连接了现有的节点。

原表 c	输出 a	输出 b	期望 a	期望 b	是否正确
空	空	空	空	空	√
7	7	空	7	空	√
1,2	1	2	1	2	√
1,2,3,4,5	1,3,5	2,4	1,3,5	2,4	√
10,20,30,40	10,30	20,40	10,30	20,40	√
-1,-1,0,7	-1,0	-1,7	-1,0	-1,7	√

2.5 测试用例截图

第一问

```
Exe=C:\mingw64\bin\gdb.exe' '--interpreter=mi'
Engine-In-1c43terw.4av' '--stdout=Microsoft-MIEngi
Exe=C:\mingw64\bin\gdb.exe' '--interpreter=mi'
c = [] → a = [], b = []
c = [7 ] → a = [7 ], b = []
c = [1 2 ] → a = [1 ], b = [2 ]
c = [1 2 3 4 5 ] → a = [1 3 5 ], b = [2 4 ]
c = [10 20 30 40 ] → a = [10 30 ], b = [20 40 ]
c = [-1 -1 0 7 ] → a = [-1 0 ], b = [-1 7 ]
PS C:\Users\Macro\Desktop\hw2>
```

第二问

```
c = [] → a = [], b = []
c = [] → a = [7 ], b = []
c = [] → a = [1 ], b = [2 ]
c = [] → a = [1 3 5 ], b = [2 4 ]
c = [] → a = [10 30 ], b = [20 40 ]
c = [] → a = [-1 0 ], b = [-1 7 ]
PS C:\Users\Macro\Desktop\hw2>
```