|  |  |
| --- | --- |
| 姓名 | 单禹嘉 |
| 学号 | 2023215177 |

|  |  |
| --- | --- |
| 实验成绩 |  |



华中师范大学计算机学院

实验报告书

课程名称：数据结构

主讲教师：沈显君

课程编号：

班级：2306

1. 实验目的

目的：

通过利用线性链表数据结构，模拟并实现两个整数集合的交集、并集和差集运算。通过实验，掌握线性链表在集合运算中的基本应用。

1. 实验基本内容

1. 需求分析： 设计一个系统，能够基于链表实现两个集合的交集、并集和差集运算。用户输入两个集合，系统输出相应运算结果。

2. 抽象数据类型定义： 定义用于表示集合的链表结构，并实现集合的基本运算，如并集、交集和差集。

3. 系统流程设计： 系统接收用户输入的集合数据，计算交集、并集、差集，输出运算结果。

4. 链表操作：

• 初始化链表（表示集合）。

• 实现并集、交集和差集的运算。

• 输出运算结果。

1. 实验环境

VSCode

1. 系统描述

该系统通过线性链表表示集合，支持对集合进行基本运算（并集、交集和差集）。每个集合中的元素都是不重复的正整数，程序会根据输入的集合内容计算集合间的交集、并集及差集，并输出结果。

1. 需求分析

1. 集合表示：

• 使用线性链表表示集合，链表中的每个节点存储一个整数。

2. 基本运算：

• 并集运算： 返回两个集合的并集，包含所有不重复的元素。

• 交集运算： 返回两个集合的交集，包含在两个集合中都存在的元素。

• 差集运算： 返回集合 A 相对集合 B 的差集，即只存在于集合 A 而不在集合 B 中的元素。

1. 详细概要设计

1）抽象数据类型定义

• 集合（Set）：

• 使用链表（List）存储集合中的元素。

struct Node {

int data;

Node\* next;

};

class Set {

private:

Node\* head;

public:

Set();

~Set();

void add(int elem); // 添加元素

void remove(int elem); // 删除元素

void display(); // 显示集合

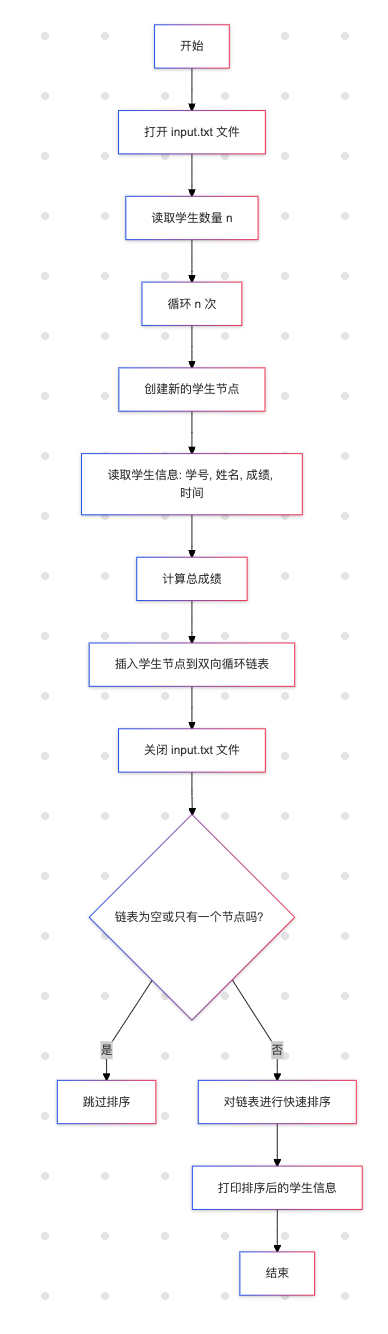
static Set Union(const Set& s1, const Set& s2); // 并集

static Set Intersection(const Set& s1, const Set& s2); // 交集

static Set Difference(const Set& s1, const Set& s2); // 差集

};

1. 系统功能模块图



3）系统函数关系调用图

• add()：添加元素到集合中。

• remove()：从集合中删除指定元素。

• Union()：计算两个集合的并集。

• Intersection()：计算两个集合的交集。

• Difference()：计算两个集合的差集。

1. 系统详细设计原理与过程

• 集合初始化： 构造空链表，用户可通过 add() 函数添加元素。

• 并集运算： 将集合 A 和集合 B 中的所有元素加入到结果集合中，去除重复元素。

• 交集运算： 遍历集合 A，检查集合 B 中是否存在相同元素，若存在则加入结果集合。

• 差集运算： 遍历集合 A，检查集合 B 中是否存在相同元素，若不存在则加入结果集合。

1. 系统运行结果与分析

假设输入两个集合 S1={3,5,6,9,12,27,35} 和 S2={2,6,8,10,12,27,31,35,43,51,55,63}。

运行结果：

• 并集： S1 ∪ S2 = {2,3,5,6,8,9,10,12,27,31,35,43,51,55,63}

• 交集： S1 ∩ S2 = {6,12,27,35}

• 差集： S1 - S2 = {3,5,9}

1. 系统使用说明

1. 启动程序后，输入两个集合的元素。

2. 系统会自动计算并输出集合的并集、交集和差集。

3. 用户可以继续输入新的集合进行运算。

1. 系统调试经验与心得

1. 实验收获：

• 掌握了利用链表实现集合运算的基本原理和方法。

• 学会了如何在链表结构中处理去重和交叉运算。

2. 改进建议：

• 可以使用平衡树或哈希表来提高集合运算的效率，特别是对于大规模数据集合时。

1. 附录（源代码）与说明

*#include* <iostream>

using namespace std;

*// 定义节点结构*

struct Node {

int data;

Node\* next;

};

*// 有序链表类*

class OrderedList {

private:

Node\* head;

public:

OrderedList() {

head = nullptr;

}

*// 析构函数*

~OrderedList() {

*while* (head != nullptr) {

Node\* temp = head;

head = head->next;

delete temp;

}

}

*// 添加元素到链表，保持有序*

void AddElem(int *elem*) {

Node\* newNode = new Node{ *elem*, nullptr};

*if* (head == nullptr || head->data > *elem*) {

newNode->next = head;

head = newNode;

} *else* {

Node\* curr = head;

*while* (curr->next != nullptr && curr->next->data < *elem*) {

curr = curr->next;

}

newNode->next = curr->next;

curr->next = newNode;

}

}

*// 删除元素*

void DelElem(int *elem*) {

*if* (head == nullptr) *return*;

*if* (head->data == *elem*) {

Node\* temp = head;

head = head->next;

delete temp;

*return*;

}

Node\* curr = head;

*while* (curr->next != nullptr && curr->next->data != *elem*) {

curr = curr->next;

}

*if* (curr->next != nullptr) {

Node\* temp = curr->next;

curr->next = curr->next->next;

delete temp;

}

}

*// 打印链表*

void Print() const {

Node\* curr = head;

*while* (curr != nullptr) {

cout << curr->data << " ";

curr = curr->next;

}

cout << endl;

}

*// 获取链表头部*

Node\* GetHead() const {

*return* head;

}

};

class Set {

private:

OrderedList list;

public:

*// 添加元素到集合*

void AddElem(int *elem*) {

list.AddElem(*elem*);

}

*// 删除集合中的元素*

void DelElem(int *elem*) {

list.DelElem(*elem*);

}

*// 并集操作*

static Set Union(const Set& *s1*, const Set& *s2*) {

Set result;

Node\* curr1 = *s1*.list.GetHead();

Node\* curr2 = *s2*.list.GetHead();

*// 合并两个有序链表*

*while* (curr1 != nullptr && curr2 != nullptr) {

*if* (curr1->data < curr2->data) {

result.AddElem(curr1->data);

curr1 = curr1->next;

} *else* *if* (curr1->data > curr2->data) {

result.AddElem(curr2->data);

curr2 = curr2->next;

} *else* {

result.AddElem(curr1->data);

curr1 = curr1->next;

curr2 = curr2->next;

}

}

*// 处理剩余元素*

*while* (curr1 != nullptr) {

result.AddElem(curr1->data);

curr1 = curr1->next;

}

*while* (curr2 != nullptr) {

result.AddElem(curr2->data);

curr2 = curr2->next;

}

*return* result;

}

*// 交集操作*

static Set Intersection(const Set& *s1*, const Set& *s2*) {

Set result;

Node\* curr1 = *s1*.list.GetHead();

Node\* curr2 = *s2*.list.GetHead();

*while* (curr1 != nullptr && curr2 != nullptr) {

*if* (curr1->data < curr2->data) {

curr1 = curr1->next;

} *else* *if* (curr1->data > curr2->data) {

curr2 = curr2->next;

} *else* {

result.AddElem(curr1->data);

curr1 = curr1->next;

curr2 = curr2->next;

}

}

*return* result;

}

*// 差集操作*

static Set Difference(const Set& *s1*, const Set& *s2*) {

Set result;

Node\* curr1 = *s1*.list.GetHead();

Node\* curr2 = *s2*.list.GetHead();

*while* (curr1 != nullptr) {

*if* (curr2 == nullptr || curr1->data < curr2->data) {

result.AddElem(curr1->data);

curr1 = curr1->next;

} *else* *if* (curr1->data > curr2->data) {

curr2 = curr2->next;

} *else* {

curr1 = curr1->next;

curr2 = curr2->next;

}

}

*return* result;

}

*// 打印集合中的元素*

void Print() const {

list.Print();

}

};

int main() {

Set set1, set2;

*// 输入集合1*

set1.AddElem(3);

set1.AddElem(5);

set1.AddElem(6);

set1.AddElem(9);

set1.AddElem(10);

set1.AddElem(12);

set1.AddElem(27);

set1.AddElem(35);

set1.Print(); *// 输出集合1*

*// 输入集合2*

set2.AddElem(5);

set2.AddElem(8);

set2.AddElem(10);

set2.AddElem(12);

set2.AddElem(27);

set2.AddElem(31);

set2.AddElem(42);

set2.AddElem(51);

set2.AddElem(55);

set2.AddElem(63);

set2.Print(); *// 输出集合2*

*// 求并集*

Set unionSet = Set::Union(set1, set2);

cout << "Union: ";

unionSet.Print();

*// 求交集*

Set intersectionSet = Set::Intersection(set1, set2);

cout << "Intersection: ";

intersectionSet.Print();

*// 求差集*

Set differenceSet = Set::Difference(set1, set2);

cout << "Difference: ";

differenceSet.Print();

*return* 0;

}