数据结构课程实践　第二章习题

张昊 1927405160

【实验 2】 用单链表实现集合的基本操作。需要注意集合中元素的唯一性，即在单链表中不存在值相同的结点，算法思路仍然是以遍历为基础。设两个单链表和 L1 和 L2 别表示两个集合，设计算法实现并、交、差、判断口 L1 是否是 L2 的子集等基本操作。(set.hpp, Lab2.cpp)

集合类声明：

```
// 继承自类模版 LinkedList 的 Set 类
template<typename Entry>
class Set : public LinkedList<Entry> {
public:
    using LinkedList<Entry>::notFound;
    // 构造函数
    Set();
    Set(Entry dataArray[], int len);
    Set(const Set<Entry> &list);
    // 覆盖 insert 方法，保证集合元素的唯一性
    void insert(int index, const Entry &elem) override;
    // 差集
    Set<Entry> set_difference(const Set<Entry> &set) const;
    // 交集
    Set<Entry> set_intersection(const Set<Entry> &set) const;
    // 并集
    Set<Entry> set_union(const Set<Entry> &set) const;
    // 判断子集
    bool is_subset(const Set<Entry> &set) const;
    // 判断超集
    bool is_superset(const Set<Entry> &set) const;
};
```

实现：

构造函数

```
template<typename Entry>
Set<Entry>::Set() : LinkedList<Entry>() {}  // 默认构造函数

template<typename Entry>
Set<Entry>::Set(Entry dataArray[], int len) {
    this->head = new Node<Entry>{Entry(), nullptr};
    this->size = len;
    for (int i = this->size - 1; i >= 0; --i) { // 查找是否已经存在
        if (this->locate(dataArray[i]) != Set<Entry>::notFound) {
            this->size--;
            continue;
        }
        auto newNode = new Node<Entry>{dataArray[i], this->head->next};
        this->head->next = newNode;
    }
}
```

```
template<typename Entry>
Set<Entry>::Set(const Set<Entry> &list)  // 复制构造函数
        : LinkedList<Entry>((const LinkedList<Entry> &) list) {}
```

调整的插入算法：

```
template<typename Entry>
void Set<Entry>::insert(int index, const Entry &elem) {
    int position = this->locate(elem);
    if (position == Set<Entry>::notFound) {  // 只有不存在该元素才插入
        LinkedList<Entry>::insert(index, elem);
        return;
    } else if (position == index) {
        return;
    }
    throw std::runtime_error{"'elem' is already in the set, position is "
                             + std::to_string(position)};
}
```

差集：

```
template<typename Entry>
Set<Entry> Set<Entry>::set_difference(const Set<Entry> &set) const {
    Set<Entry> result;
    Node<Entry> *ptr = this->head->next;
    while (ptr != nullptr) {
        if (set.locate(ptr->data) == Set<Entry>::notFound) {    // 建立差集
            result.insert(1, ptr->data);
        }
        ptr = ptr->next;
    }
    return result;
}
```

交集：

```
template<typename Entry>
Set<Entry> Set<Entry>::set_intersection(const Set<Entry> &set) const {
    Set<Entry> result;
    Node<Entry> *ptr = this->head->next;
    while (ptr != nullptr) {
        if (set.locate(ptr->data) != Set<Entry>::notFound) {    // 建立交集
            result.insert(1, ptr->data);
        }
        ptr = ptr->next;
    }
    return result;
}
```

并集：

```
template<typename Entry>
Set<Entry> Set<Entry>::set_union(const Set<Entry> &set) const {
    Set<Entry> result(*this);
    Node<Entry> *ptr = set.head->next;
    while (ptr != nullptr) {
        if (result.locate(ptr->data) == Set<Entry>::notFound) { // 建立并集
            result.insert(1, ptr->data);
        }
        ptr = ptr->next;
    }
    return result;
}
```

判断子集、超集：

```
template<typename Entry>
bool Set<Entry>::is_subset(const Set<Entry> &set) const {
    Node<Entry> *ptr = this->head->next;
    while (ptr != nullptr) {
        // 判断子集
        if (set.locate(ptr->data) == Set<Entry>::notFound) {
            return false;
        }
        ptr = ptr->next;
    }
    return true;
}

template<typename Entry>
bool Set<Entry>::is_superset(const Set<Entry> &set) const {
    return set.is_subset(*this);  // 判断超集
}
```

【实验 3】 在单链表的具体应用中，数据元素有了具体的数据类型，因此，需要根据个单链题目设计结点的存储结构。某商店的仓库中，对电视机按其价格从低到高建表，链表的每个结点指出同样价格的电视机的台数、现有 m 台价格为 n 元的电视机入库，请应用单链表类完成仓库的进货管理。(warehouse.h, warehouse.cpp, Lab3.cpp)

结点（货物）数据存储结构：

```
struct Goods {
    double price;
    int count;
    bool operator==(const Goods &goods) const;
};
bool Goods::operator==(const Goods &goods) const {
    return std::abs(price - goods.price) < 1e-9;
}
```

仓库类继承自单链表类模板，指定模板参数为 Goods：

```
class Warehouse : LinkedList<Goods> {
public:
    Warehouse() : LinkedList<Goods>() {}    // 构造函数
    void deliver(const Goods &goods);   // 进货管理
    std::string to_string() const;  // 遍历仓库，转换为字符串
};
```

进货方法实现：

```
void Warehouse::deliver(const Goods &goods) {
    if (empty()) {  // 空链表需单独处理
        head->next = new Node<Goods>{goods, head->next};
        ++size;
        return;
    }
    Node<Goods> *pre = head, *next = head->next;
    // 保证按照从小到大顺序插入
    while (next != nullptr) {
        if (next->data.price > goods.price) {
            pre->next = new Node<Goods>{goods, pre->next};
            ++size;
            return;
        } else if (next->data == goods) {  // 已经存在则更新数量
            next->data.count += goods.count;
            return;
        }
        pre = pre->next;
        next = next->next;
    }
    pre->next = new Node<Goods>{goods, pre->next};
    ++size;
}
```

遍历算法实现：

```
std::string Warehouse::to_string() const {
    std::ostringstream out;
    Node<Goods> *ptr = head->next;
    out << "Warehouse: { ";
    while (ptr != nullptr) {
        out << "$" << ptr->data.price << ": " << ptr->data.count;
        ptr = ptr->next;
        if (ptr != nullptr) { out << ", "; }
    }
    out << " }";  return out.str();
}
```