数据结构作业 (2)

1. 简述下列术语：线性表，顺序表，链表。

- 线性表：线性表是由$n(n \geq 0)$个数据元素（结点）组成的有限序列。该序列中的所有结点具有相同的数据类型。
- 顺序表：顺序表指的是用一组地址连续的存储单元依次存储数据元素的线性表。其中，逻辑上相邻的元素在物理位置上也相邻。
- 链表：链表指的是用一组任意的存储单元存储数据元素的线性表。

2. 何时选用顺序表，何时选用链表作为线性表的存储结构合适？各自的主要优缺点是什么？

- 对于顺序表：
    - **优点**：顺序表能够随机访问存取表中的任一元素，它的存储位置可用一个简单、直观的公式来表示
    - **缺点**：顺序表的容量是固定的，在使用顺序表时需要预估数据量大小，若表不满，会造成空间浪费；若表满，又需要重新申请空间，进行数据迁移，增加了时间开销。并且，顺序表的插入和删除操作需要移动大量的元素，效率较低。
    - **适用场景**：顺序表适用于数据量固定，对数据的随机访问较多的场景。
- 对于链表：
    - **优点**：链表的容量时动态的，能够根据数据量的变化灵活进行扩容或缩容，不会造成空间浪费。并且，链表的插入和删除操作只需要修改指针，效率较高。
    - **缺点**：链表的随机访问效率较低，需要按照顺序逐个进行查找，效率较低。
    - **适用场景**：链表适用于数据量不固定，对数据的随机访问较少的场景。

3. 在顺序表中插入和删除一个结点平均需要移动多少个结点？具体的移动次数取决于哪两个因素？

- 在顺序表中插入和删除一个结点平均需要移动$\frac{n}{2}$个结点。
- 其移动次数取决于插入或删除的位置和表中元素的个数。
    - 若在表的最后插入或删除一个结点，只需要移动一个结点。
    - 若在表的最前面插入或删除一个结点，需要移动n个结点。
    - 若在表的中间插入或删除一个结点，需要移动$\frac{n}{2}$个结点。

4. 链表所表示的元素是否有序？如有序，则有序性体现于何处？链表所表示的元素是否一定要在物理上是相邻的？有序表的有序性又如何理解？

- 链表所表示的元素是有序的。其有序性体现在链表的指针域上。链表是通过每个结点的指针域将线性表的n个结点按其逻辑次序链接在一起的。
- 链表所表示的元素不一定要在物理上是相邻的。在链表中，任何两个元素之间的存储位置之间没有固定的联系，然而，每个元素的存储位置都包含在其相邻元素的信息之中。
- 有序表的有序性指的是逻辑上的有序性，意味着表中的元素按照某种规则排列。

5. 设顺序表L是递增有序表，试写一算法，将x插入到L中并使L仍是递增有序表。

- 伪代码如下：

```
void OrderInsert(Node & list, Comparable & data) {
    int len = list.length;
    for (int i = 0; i < len; ++i) {
        if (list[i] > data) {
            for (int j = len; j > i; --j) {
                list[j] = list[j - 1];
            }
            list[i] = data;
            break;
        }
    }
}
```

6. 写一求单链表的结点数目ListLength(L)的算法。

7. 写一算法将单链表中值重复的结点删除，使所得的结果链表中所有结点的值均不相同。

8. 写一算法从一给定的向量A删除值在x到y（x<=y）之间的所有元素（注意：x和y是给定的参数，可以和表中的元素相同，也可以不同）。

9. 设A和B是两个元素值递增有序的单链表，写一算法将A和B归并为按元素值递减的有序的单链表C，试分析算法的时间复杂度。

- 代码如下，其中Merge函数实现9所要求的算法，其时间复杂度为$O(n)$。

6、7、8、9题的代码实现如下：

```cpp
template <typename Comparable>
struct ListNode {
    Comparable data;
    ListNode *next;
};


template <typename Comparable>
struct HeadNode : public ListNode<Comparable> {};


template <typename Comparable>
class List {
public: // public values
    int size;
    HeadNode<Comparable> *head;

public: // constructor and destructor
    List() {
        head = new HeadNode<Comparable>;
        head->next = nullptr;
        size = 0;
    }
    ~List() {
        ListNode<Comparable> *preNode, curNode;
        preNode = head; curNode = head->next;
        while (curNode != nullptr) {
            delete preNode;
            preNode = curNode;
            curNode = curNode->next;
        }
        delete preNode;
    }

public: // public methods
    int ListLength() const { return size; }
    int ListLength(int) const;
    void OrderInsert(const Comparable &x);
    void DelRepeat();
    void DelReapeat(int);
    void DelRange(const Comparable &min, const Comparable &max);

public: // static methods
    static HeadNode<Comparable> * Merge(List<Comparable> &list1, List<Comparable> &list2);
};
```

```cpp
template <typename Comparable>
void List<Comparable>::OrderInsert(const Comparable &x) {
    ListNode<Comparable> *preNode, *curNode;
    preNode = head; curNode = head->next;
    while (curNode != nullptr && curNode->data < x) {
        preNode = curNode;
        curNode = curNode->next;
    }
    ListNode<Comparable> *newNode = new ListNode<Comparable>;
    newNode->data = x;
    newNode->next = curNode;
    preNode->next = newNode;
    ++size;
}


template <typename Comparable>
void List<Comparable>::ListLength(int) const { // if you dont know the length
    ListNode<Comparable> *curNode;
    curNode = head->next;
    int count = 0;
    while (curNode != nullptr) {
        ++count;
        curNode = curNode->next;
    }
    return count;
}


template <typename Comparable>
void List<Comparable>::DelRepeat() {
    // if the list is increamentally ordered
    ListNode<Comparable> *preNode, *curNode;
    preNode = head; curNode = head->next;
    while (curNode != nullptr) {
        if (curNode->data == preNode->data) {
            preNode->next = curNode->next;
            delete curNode;
            curNode = preNode->next;
            --size;
        } else {
            preNode = curNode;
            curNode = curNode->next;
        }
    }
```

```
}


template <typename Comparable>
void List<Comparable>::DelReapeat(int) {
    // if the list is randomly ordered
    ListNode<Comparable> *preNode, *curNode;
    preNode = head; curNode = head->next;
    Comparable *table = new Comparable[size];
    int count = 0;

    auto checkRepeat = [&table, &count](const Comparable &x) {
        for (int i = 0; i < count; ++i)
            if (table[i] == x) return true;
        return false;
    };

    while (curNode != nullptr) {
        if (checkRepeat(curNode->data)) {
            preNode->next = curNode->next;
            delete curNode;
            curNode = preNode->next;
            --size;
        } else {
            table[count++] = curNode->data;
            preNode = curNode;
            curNode = curNode->next;
        }
    }

    delete [] table;
}


template <typename Comparable>
void List<Comparable>::DelRange(const Comparable &min, const Comparable &max) {
    ListNode<Comparable> *preNode, *curNode;
    preNode = head; curNode = head->next;
    while (curNode != nullptr) {
        if (curNode->data >= min && curNode->data <= max) {
            preNode->next = curNode->next;
            delete curNode;
            curNode = preNode->next;
            --size;
        } else {
            preNode = curNode;
```

```
            curNode = curNode->next;
        }
    }
}


template <typename Comparable>
HeadNode<Comparable> * List<Comparable>::Merge(List<Comparable> &list1, List<Comparable> &list2) {
    HeadNode<Comparable> *head1, *head2, *head3;
    head1 = list1.head; head2 = list2.head;
    head3 = new HeadNode<Comparable>;
    head3->next = nullptr;

    ListNode<Comparable> *p1, *p2;
    p1 = head1->next; p2 = head2->next;
    while (p1 != nullptr || p2 != nullptr) {
        if (p1->data < p2->data) {
            ListNode<Comparable> *newNode = new ListNode<Comparable>;
            newNode->data = p1->data;
            newNode->next = head3->next;
            head3->next = newNode;
            p1 = p1->next;
        } else {
            ListNode<Comparable> *newNode = new ListNode<Comparable>;
            newNode->data = p2->data;
            newNode->next = head3->next;
            head3->next = newNode;
            p2 = p2->next;
        }
    }
    if (p1 != nullptr) {
        ListNode<Comparable> *newNode = new ListNode<Comparable>;
        newNode->data = p1->data;
        newNode->next = head3->next;
        head3->next = newNode;
        p1 = p1->next;
    } else if (p2 != nullptr) {
        ListNode<Comparable> *newNode = new ListNode<Comparable>;
        newNode->data = p2->data;
        newNode->next = head3->next;
        head3->next = newNode;
        p2 = p2->next;
    }

    return head3;
}
```