线性表

本文档需要用到的编程相关知识点：

(1) C语言：struct（结构体）、typedef、malloc（free）、函数做参数。

(2) C++：抽象类、模板类、new（delete）、运算符重载。

本文档的参考教材：

(1) 数据结构（C语言版） 严蔚敏

(2) 数据结构与算法分析（C++版） Clifford A.Shaffer

一. C语言实现

1. 顺序表的结构及相关算法

顺序表的实现参照教材19页。（由于程序引入了 C++ 引用传参的方式，故程序请建立为.cpp格式）

```
// 定义顺序表的结构及相关算法
#include <stdio.h>
#include <stdlib.h>

#define OK 1
#define OVERFLOW -1
#define LIST_INIT_SIZE 2 // 线性表存储空间初始分配容量
#define LISTINCREMENT 10 // 线性表存储空间分配增量
#define TRUE 1
#define FALSE 0
#define NOTEXIST 0
#define ERROR 0

typedef int Status;
typedef int ElemType;

struct Sqlist {      //顺序表类型
    ElemType *elem; // 存储空间基址
    int length;     // 当前长度
    int listsize;   // 当前分配的存储容量
};

Status InitList_Sq(struct Sqlist &L)
{
    // 构造一个空的线性表L
    L.elem = (ElemType *)malloc(LIST_INIT_SIZE * sizeof(struct Sqlist));
    if (!L.elem)
        return (OVERFLOW);          // 存储分配失败
    L.length = 0;                   // 空表长度为0
    L.listsize = LIST_INIT_SIZE;  // 初始存储容量
    return OK;
}  // InitList_Sq
```

```
33
34  void DestoryList_Sq(struct Sqlist &L)
35  {
36      // 销毁线性表L
37      // 要求线性表L存在
38      free(L.elem);
39      L.elem = NULL;
40  }  // DestoryList_Sq
41
42  void ClearList_Sq(struct Sqlist &L)
43  {
44      // 将L重置为空表
45      // 要求线性表L存在
46      L.length = 0;
47  }  // ClearList_Sq
48
49  Status ListEmpty_Sq(struct Sqlist L)
50  {
51      // 若L为空表，返回TRUE，否则返回FALSE
52      // 要求线性表L存在
53      if (0 == L.length) {
54          return TRUE;
55      } else {
56          return FALSE;
57      }
58  }  // ListEmpty_Sq
59
60  Status ListLength_Sq(struct Sqlist L)
61  {
62      // 要求线性表已存在
63      // 返回L中数据元素个数
64      return L.length;
65  }  // ListLength_Sq
66
67  void GetElem_Sq(struct Sqlist L, int i, ElemType &e)
68  {
69      // 要求线性表存在，1<=i<=ListLength_Sq(L)
70      // e返回L中第i个数据元素的值
71      e = *(L.elem + i - 1);
72  }  // GetElem_Sq
73
74  Status compare_equal_Sq(ElemType e1, ElemType e2)
75  {
76      // 判断两个数据元素是否相等，相等返回TRUE，不等返回FALSE
77      if (e1 == e2)
78          return TRUE;
79      else
80          return FALSE;
81  }  // compare_equal_Sq
82
83  Status LocateElem_Sq(struct Sqlist L, ElemType e,
84                       Status (*compare)(ElemType, ElemType))
85  {
86      // 线性表L已存在，compare()是数据元素判定函数
87      // 返回L中第一个与e满足关系compare()的数据元素的位序。若这样的数据元素不存在，返回
    0
88      ElemType *p = L.elem;
89      int i;  // 位序
```

```
    for (i = 1; i <= L.length; i++) {
        if (compare(e, *(p + i - 1))) {
            return i;
        }
    }
    return 0;
}  // LocateElem_Sq

Status PriorElem_Sq(struct Sqlist L, ElemType cur_e)
{
    // 要求线性表L存在
    // 若cur_e是L的数据元素，且不是第一个，则返回它的前驱，否则操作失败
    int pos;
    pos = LocateElem_Sq(L, cur_e, compare_equal_Sq);
    if (!pos || 1 == pos)  // 元素cur_e在L中不存在或为第一个
        return NOTEXIST;
    else
        return *(L.elem + pos - 2);
}  // PriorElem_Sq

Status NextElem_Sq(struct Sqlist L, ElemType cur_e)
{
    // 要求线性表L存在
    // 若cur_e是L的数据元素，且不是最后一个，则返回它的后驱，否则操作失败
    int pos;
    pos = LocateElem_Sq(L, cur_e, compare_equal_Sq);
    if (!pos || L.length == pos)  // 元素cur_e在L中不存在或为最后一个
        return NOTEXIST;
    else
        return *(L.elem + pos);
}  // NextElem_Sq

void ListInsert_Sq(struct Sqlist &L, int i, ElemType e)
{
    // 要求线性表存在，1<=i<=ListLength_Sq(L)+1
    // 在第i个位置之前插入数据元素e，L的长度加1
    int j;
    ElemType *newbase;
    if (L.length + 1 > L.listsize) {
        newbase = (ElemType *)realloc(
            L.elem, (L.listsize + LISTINCREMENT) * sizeof(ElemType));
        if (!newbase)
            exit(OVERFLOW);           // 存储分配失败
        L.elem = newbase;             // 新基址
        L.listsize += LISTINCREMENT;  // 增加存储容量
    }
    for (j = L.length; j >= i; j--) {
        *(L.elem + j) = *(L.elem + j - 1);
    }
    *(L.elem + i - 1) = e;
    L.length++;  // 表长加1
}  // ListInsert_Sq

void ListDelete_Sq(struct Sqlist &L, int i, ElemType &e)
{
    // 线性表存在且非空，1<=i<=ListLength_Sq(L)
    // 删除L的第i个元素，并用e返回其值，L的长度减1
    int j;
```

```
    e = *(L.elem + i - 1);
    for (j = i; j < L.length; j++) {
        *(L.elem + j - 1) = *(L.elem + j);
    }
    L.length--;
}  // ListDelete_Sq

Status ListTraverse_Sq(struct Sqlist L, Status (*visit)(ElemType))
{
    // 要求线性表存在
    // 依次对L的每个元素调用函数visit(),
    // 一旦visit()失败，则操作失败，返回FALSE,否则返回TRUE
    int i;
    for (i = 0; i < L.length; i++) {
        if (!visit(*(L.elem + i)))
            return FALSE;
    }
    return TRUE;
}  // ListTraverse_Sq

Status visit(ElemType e)
{
    printf("%d ", e);
    return TRUE;
}  // visit_display_Sq

// 算法

// 算法2.1
void UnionList_Sq(struct Sqlist &La, struct Sqlist Lb)
{
    // 将Lb中存在但La中不存在的元素插入到La中
    int La_len = ListLength_Sq(La);
    int Lb_len = ListLength_Sq(Lb);
    int i;
    ElemType e;
    for (i = 1; i <= Lb_len; i++) {
        GetElem_Sq(Lb, i, e);  // 取第i个元素赋给e
        if (!LocateElem_Sq(La, e, compare_equal_Sq)) {
            ListInsert_Sq(La, ++La_len, e);
        }
    }
}  // UnionList_Sq

// 算法2.2
void MergeList_Sq(struct Sqlist La, struct Sqlist Lb, struct Sqlist &Lc)
{
    // 已知线性表La和Lb中数据元素按值非递减排列
    // 归并La和Lb得到新的线性表Lc,Lc的数据元素也按值非递减排列
    int i, j, k, ai, bj;
    int La_len, Lb_len;
    ElemType e;
    InitList_Sq(Lc);
    i = j = k = 1;
    La_len = ListLength_Sq(La);
    Lb_len = ListLength_Sq(Lb);
    while (i <= La_len && j <= Lb_len) {
        // La, Lb为非空表
```

```
        GetElem_Sq(La, i, ai);
        GetElem_Sq(Lb, j, bj);
        if (ai <= bj) {
            ListInsert_Sq(Lc, k++, ai);
            ++i;
        } else {
            ListInsert_Sq(Lc, k++, bj);
            ++j;
        }
    }
    while (i <= La_len) {
        GetElem_Sq(La, i++, ai);
        ListInsert_Sq(Lc, k++, ai);
    }
    while (j <= Lb_len) {
        GetElem_Sq(Lb, j++, bj);
        ListInsert_Sq(Lc, k++, bj);
    }
}  // MergeList
```

顺序表测试程序：（测试程序代码放在实现代码下方即可）

```
int main()
{
    int arr[8] = {33, 12, 75, 0, 49, 67, 8, 999};
    Sqlist list1;

    InitList_Sq(list1);
    for (int i = 0; i < 8; i++)  // 将arr数组中的8个元素插入顺序表中
        ListInsert_Sq(list1, i + 1, arr[i]);

    printf("There are %d elements in sequence table list1.\n",
           ListLength_Sq(list1));
    printf("The %d elements are : \n", ListLength_Sq(list1));
    ListTraverse_Sq(list1, visit);  // 遍历顺序表中的元素

    // 返回第6个元素的前驱与后继
    printf("\n\nThe previous element of the sixth element of list1 is: %d",
           PriorElem_Sq(list1, 67));
    printf("\nThe next element of the sixth element of list1 is: %d",
           NextElem_Sq(list1, 67));

    int e = 0;
    ListDelete_Sq(list1, 3, e);  // 删除顺序表中的第三个元素
    printf(
        "\n\nThe remaining elements in the sequence table after deleting
the "
        "third element are: \n");
    ListTraverse_Sq(list1, visit);

    DestoryList_Sq(list1);  // 销毁顺序表

    return 0;
}
```

测试程序运行结果：

There are 8 elements in sequence table list1.
The 8 elements are :
33 12 75 0 49 67 8 999

The previous element of the sixth element of list1 is: 49
The next element of the sixth element of list1 is: 8

The remaining elements in the sequence table after deleting the third element are:
33 12 0 49 67 8 999

---

2. 线性链表的结构及相关算法

链表的实现有两种：(1) 不带头结点的线性链表（教材28页）。(2) 带头结点的线性链表（教材37页）。本文档参照第二种。

```
//带头结点的线性链表的构造及相关算法
#include <stdio.h>
#include <stdlib.h>

#define OK 1
#define OVERFLOW -1
#define TRUE 1
#define FALSE 0
#define ERROR 0

typedef int Status;
typedef int ElemType;

typedef struct LNode2 {  // 结点类型
    ElemType data;
    struct LNode2 *next;
} * Link, *Position;

typedef struct {       // 链表类型
    Link head, tail;  // 分别指向线性链表中的头结点和最后一个结点
    int len;          // 指示线性表中数据元素的个数
} LinkList;


Status MakeNode(Link &p, ElemType e)
{
    // 分配由p指向的值为e的结点，并返回OK；若分配失败，则返回ERROR
    p = (Link)malloc(sizeof(LNode2));
    if (!p)  // 分配失败
        return ERROR;
    p->data = e;
    return OK;
}  // MakeNode

void FreeNode(Link &p)
{
```

```
    // 释放p所指结点
    free(p);
    p = NULL;
}  // FreeNode

Status InitList(LinkList &L)
{
    // 构造一个空的线性链表L
    Link p;
    p = (Link)malloc(sizeof(LNode2));  // 生成头结点
    if (!p)
        return ERROR;
    p->next = NULL;  // 头结点的下一个元素置空
    L.head = L.tail = p;
    L.len = 0;  // 长度初始为0

    return OK;
}  // InitList

Status ClearList(LinkList &L)
{
    // 将L置为空表，并释放原链表的结点空间
    Link p, q;
    p = L.head->next;
    L.head->next = NULL;
    while (NULL != p) {  // 遍历并释放所有节点
        q = p->next;
        free(p);
        p = q;
    }
    L.tail = L.head;
    L.len = 0;
    return OK;
}  // ClearList

Status DestroyList(LinkList &L)
{
    // 销毁线性表L，L不再存在
    ClearList(L);  // 释放除头尾结点的所有结点
    free(L.head);  // 释放头尾结点
    L.head = L.tail = NULL;
    L.len = 0;
    return OK;
}  // DestroyList

Status InsFirst(Link h, Link s)
{
    // 已知h指向链表头结点，将s所指结点插入在第一个结点之前
    s->next = h->next;
    h->next = s;
    return OK;
}  // InsFirst

Status DelFirst(Link h, Link &q)
{
    // 已知h指向链表头结点，删除第一个结点并以q返回，链表为空返回ERROR
    if (NULL == h->next)  // 链表为空
        return ERROR;
```

```
    q = h->next;
    h->next = q->next;
    // TODO: q->next = NULL;是否需要?
    q->next = NULL;
    return OK;
}  // DelFirst

Status Append(LinkList &L, Link s)
{
    // 将指针s所指的一串结点链接在L的最后一个结点之后，并修改尾指针
    int i = 1;                  // 计数，记录s链接的结点数
    L.tail->next = s;           // 链接
    while (NULL != s->next) {  // 寻找尾指针
        s = s->next;
        i++;
    }
    L.tail = s;  // 修改尾指针
    L.len += i;
    return OK;
}  // Append

Status Remove(LinkList &L, Link &q)
{
    // 删除L尾结点，并以q返回
    Link p;
    p = L.head;
    if (L.head == L.tail)
        return ERROR;
    q = L.tail;
    while (L.tail != p->next) {  // 寻找尾指针
        p = p->next;
    }
    p->next = NULL;  // 删除尾指针结点
    L.tail = p;      // 修改尾指针
    L.len--;
    return OK;
}  // Remove

Status InsBefore(LinkList &L, Link &p, Link s)
{
    // 已知p指向线性链表L中的一个结点，将s所指结点插入在p所指结点之前，
    // 并修改指针p指向新插入的结点
    Link h = L.head;
    while (p != h->next) {  // 搜索p所指结点
        h = h->next;
    }
    h->next = s;
    s->next = p;
    p = s;
    L.len++;
    return OK;
}  // InsBefore

Status InsAfter(LinkList &L, Link &p, Link s)
{
    // 已知p指向线性链表L中的一个结点，将s所指结点插入在p所指结点之后，
    // 并修改指针p指向新插入的结点
    s->next = p->next;
```

```
    p->next = s;
    if (p == L.tail)
        L.tail = s;  // 修改尾指针
    p = s;
    L.len++;
    return OK;
}  // InsAfter

Status SetCurElem(Link &p, ElemType e)
{
    // 已知p指向线性链表中的一个结点，用e更新p所指结点中数据元素的值
    p->data = e;
    return OK;
}  // SetCurElem

ElemType GetCurElem(Link p)
{
    // 已知p指向表中一个结点，返回p所指结点中数据元素的值
    return p->data;
}  // GetCurElem

Status ListEmpty(LinkList L)
{
    // L为空表返回TRUE，否则FALSE
    if (NULL == L.head->next)
        return TRUE;
    else
        return FALSE;
}  // ListEmpty

int ListLength(LinkList L)
{
    // 返回L中元素个数
    return L.len;
}  // ListLength

Position GetHead(LinkList L)
{
    // 返回L中头结点位置
    return L.head;
}  // GetHead

Position GetLast(LinkList L)
{
    //返回L中尾结点位置
    return L.tail;
}  // GetLast

Position PriorPos(LinkList L, Link p)
{
    // 已知p指向线性链表L中的一个结点，返回p所指结点的直接前驱的位置
    // 若无前驱，则返回NULL
    Link pri = L.head;
    while (pri->next != p) {
        pri = pri->next;
    }
    if (L.head == pri)
        return NULL;  // 没有直接前驱
```

```
    return pri;
}  // PriorPos

Position NextPos(LinkList L, Link p)
{
    // 返回后继位置，无则返回NULL
    if (L.tail == p)
        return NULL;
    return p->next;
}  // NextPos

Status LocatePos(LinkList L, int i, Link &p)
{
    //返回p指示线性链表L中第i个结点的位置并返回OK，i值不合法时返回ERROR
    // i=0为头结点
    if (i < 0 || i > L.len)
        return ERROR;
    p = L.head;
    while (0 != i) {
        i--;
        p = p->next;
    }
    return OK;
}  // LocatePos

Position LocateElem(LinkList L, ElemType e, int (*compare)(ElemType,
ElemType))
{
    // 返回线性链表L中第1个与e满足函数compare()判定关系的元素的位置，
    // 若不存在这样的元素，则返回NULL
    int i;
    Link p;
    p = L.head->next;  // 指向第一个结点
    for (i = 0; i < L.len; i++) {
        if (0 == (*compare)(e, p->data))
            return p;
        p = p->next;
    }
    return NULL;
}  // LocateElem

int compare(ElemType e1, ElemType e2)
{  // 比较函数
    if (e1 > e2)
        return 1;
    else if (e1 == e2)
        return 0;
    else
        return -1;
}

Status ListTraverse(LinkList L, void (*visit)(Link))
{
    // 依次对L的每个数据元素调用函数visit()
    Link p = L.head->next;
    while (NULL != p) {  // 遍历所有元素
        visit(p);
        p = p->next;
```

```
    }
    return OK;
}  // ListTraverse

void visit(Link p) { printf("%d ", p->data); }  // visit

/**
 * 算法2.20，在带头结点的单链线性表L的第i个元素之前插入元素e
 */
Status ListInsert(LinkList &L, int i, ElemType e)
{
    Link h, s;
    if (!LocatePos(L, i - 1, h))
        return ERROR;
    if (!MakeNode(s, e))
        return ERROR;
    if (h == L.tail)
        L.tail = s;  // 修改尾指针
    InsFirst(h, s);  // 对于从第不个结点开始的链表，第i-1个结点是它的头结点
    L.len++;
    return OK;
}

/**
 * 算法2.21
 */
Status MergeList(LinkList &La, LinkList &Lb, LinkList &Lc,
                 int (*compare)(ElemType, ElemType))
{
    Link ha, hb, pa, pb, q;
    ElemType a, b;
    if (!InitList(Lc))
        return ERROR;
    ha = GetHead(La);
    hb = GetHead(Lb);
    pa = NextPos(La, ha);
    pb = NextPos(Lb, hb);
    while (pa && pb) {
        a = GetCurElem(pa);
        b = GetCurElem(pb);
        if (compare(a, b) <= 0) {  // a<=b
            DelFirst(ha, q);
            Append(Lc, q);
            pa = NextPos(La, ha);
        } else {  // a>b
            DelFirst(hb, q);
            Append(Lc, q);
            pb = NextPos(Lb, hb);
        }
    }  // while
    if (pa)
        Append(Lc, pa);
    else
        Append(Lc, pb);
    FreeNode(ha);
    FreeNode(hb);
    return OK;
}
```

二. C++实现

1. 顺序表的实现

顺序表的实现参照参考教材63页。

```
#include <iostream>

using namespace std;
void Assert(bool val, string s)
{                // If "val" is false, print a message and terminate the
program
    if (!val) {  // Assertion failed -- close the program
        cout << "Assertion Failed: " << s << endl;
        exit(-1);
    }
}

template <typename E>
class List  // List ADT
{
private:
    void operator=(const List&) {}  // Protect assignment
    List(const List&) {}            // Protect copy constructor

public:
    List() {}           // Default constructor
    virtual ~List() {}  // Base destructor

    virtual void clear() = 0;

    // Insert item at the current location
    virtual void insert(const E& item) = 0;

    // Append item at the end of the list
    virtual void append(const E& item) = 0;

    // Remove and return the current element
    virtual E remove() = 0;

    // Set the current position to the start of the list
    virtual void moveToStart() = 0;

    // Set the current position to the end of the list
    virtual void moveToEnd() = 0;

    // Move the current position one step left
    // No change if already at beginning
    virtual void prev() = 0;

    // Move the current position one step right
    // No change if already at end
    virtual void next() = 0;

    // Return the number of elements in the list
    virtual int length() const = 0;
```

```
50
51      // Return the position of the current element
52      virtual int currPos() const = 0;
53
54      // Set current position
55      virtual void moveToPos(int pos) = 0;
56
57      // Return the current element
58      virtual const E& getValue() const = 0;
59  };
60
61  template <typename E>  // Array-based list implementation
62  class Alist : public List<E>
63  {
64  private:
65      int maxSize;
66      int listSize;
67      int curr;
68      E* listArray;
69
70  public:
71      Alist(int size = 5)
72      {
73          maxSize = size;
74          listSize = curr = 0;
75          listArray = new E[maxSize];
76      }
77
78      ~Alist() { delete[] listArray; }
79
80      void clear()
81      {
82          delete[] listArray;
83          listSize = curr = 0;
84          listArray = new E[maxSize];
85      }
86
87      void insert(const int& it)
88      {
89          Assert(listSize < maxSize, "List capacity exceeded");
90          for (int i = listSize; i > curr; i--)
91              listArray[i] = listArray[i - 1];
92          listArray[curr] = it;
93          listSize++;
94      }
95
96      void append(const E& it)
97      {
98          Assert(listSize < maxSize, "List capacity exceeded");
99          listArray[listSize++] = it;
100     }
101
102     int remove()
103     {
104         Assert((curr >= 0) && (curr < listSize), "No element");
105         E it = listArray[curr];
106         for (int i = curr; i < listSize - 1; i++)
107             listArray[i] = listArray[i + 1];
```

```
        listSize--;
        return it;
    }

    void moveToStart() { curr = 0; }
    void moveToEnd() { curr = listSize; }
    void prev()
    {
        if (curr != 0)
            curr--;
    }
    void next()
    {
        if (curr < listSize)
            curr++;
    }

    int length() const { return listSize; }
    int currPos() const { return curr; }

    void moveToPos(int pos)
    {
        Assert((pos >= 0) && (pos <= listSize), "Pos out of range");
        curr = pos;
    }

    const E& getValue() const
    {
        Assert((curr >= 0) && (curr < listSize), "No current element");
        return listArray[curr];
    }
};
```

测试程序问题描述：

在已实现顺序表的基础上，编写一个函数，用于合并两个顺序表。输入的顺序表按照其元素从小到大排序，输出的顺序表要求按照元素从大到小排序。

测试程序代码：（测试程序代码放在实现代码下方即可）

```
template <typename E>
void listUnion(Alist<E>& a, Alist<E>& b, Alist<E>& c)
{
    a.moveToStart();
    b.moveToStart();
    while (a.currPos() != a.length() && b.currPos() != b.length()) {
        int valueA = a.getValue();
        int valueB = b.getValue();
        if (valueA == valueB) {  // 若a b相同，则插入c中
            c.insert(valueA);
            c.moveToStart();  // 为了保证倒序
            a.next();
            b.next();
        } else if (valueA > valueB) {  // 若b小，则将b的值插入c中，b后移，a不变
```

```
            c.insert(valueB);
            c.moveToStart();
            b.next();
        } else {  // 若a小，则将b的值插入c中，a后移，b不变
            c.insert(valueA);
            c.moveToStart();
            a.next();
        }
    }
    while (a.currPos() != a.length()) {  // 复制剩余的
        c.insert(a.getValue());
        c.moveToStart();
        a.next();
    }
    while (b.currPos() != b.length()) {  // 复制剩余的
        c.insert(b.getValue());
        c.moveToStart();
        b.next();
    }
}

int main()
{
    int m, n;
    cout << "Input m and n: ";  // m、n分别为顺序表La和Lb的容量
    cin >> m >> n;
    Alist<int> La(m);
    Alist<int> Lb(n);
    Alist<int> Lc(m + n);

    cout << "Input the elements of La: ";
    for (int i = 0; i < m; i++) {
        int element;
        cin >> element;
        La.insert(element);
        La.next();
    }
    cout << "Input the elements of Lb: ";
    for (int i = 0; i < n; i++) {
        int element;
        cin >> element;
        Lb.insert(element);
        Lb.next();
    }
    listUnion(La, Lb, Lc);

    cout << "The elements of La are: " << endl;
    La.moveToStart();
    for (int i = 0; i < m; i++) {
        cout << La.getValue() << " ";
        La.next();
    }
    cout << endl << "The elements of Lb are: " << endl;
    Lb.moveToStart();
    for (int i = 0; i < n; i++) {
        cout << Lb.getValue() << " ";
        Lb.next();
    }
```

```
    cout << endl << "The elements of Lc are: " << endl;
    Lc.moveToStart();
    for (int i = 0; i < m + n; i++) {
        cout << Lc.getValue() << " ";
        Lc.next();
    }
    system("pause");
    return 0;
}
```

测试程序运行结果：

Input m and n: 4 4
Input the elements of La: 1 3 5 7
Input the elements of Lb: 2 4 6 8
The elements of La are:
1 3 5 7
The elements of Lb are:
2 4 6 8
The elements of Lc are:
8 7 6 5 4 3 2 1

---

2. 链表的实现

单链表的实现参照参考教材65页。

```
#include <iostream>

using namespace std;
void Assert(bool val, string s)
{                // If "val" is false, print a message and terminate the
program
    if (!val) {  // Assertion failed -- close the program
        cout << "Assertion Failed: " << s << endl;
        exit(-1);
    }
}

template <typename E>
class List  // List ADT
{
private:
    void operator=(const List&) {}  // Protect assignment
    List(const List&) {}            // Protect copy constructor

public:
    List() {}           // Default constructor
    virtual ~List() {}  // Base destructor

    virtual void clear() = 0;

    // Insert item at the current location
```

```
26      virtual void insert(const E& item) = 0;
27
28      // Append item at the end of the list
29      virtual void append(const E& item) = 0;
30
31      // Remove and return the current element
32      virtual E remove() = 0;
33
34      // Set the current position to the start of the list
35      virtual void moveToStart() = 0;
36
37      // Set the current position to the end of the list
38      virtual void moveToEnd() = 0;
39
40      // Move the current position one step left
41      // No change if already at beginning
42      virtual void prev() = 0;
43
44      // Move the current position one step right
45      // No change if already at end
46      virtual void next() = 0;
47
48      // Return the number of elements in the list
49      virtual int length() const = 0;
50
51      // Return the position of the current element
52      virtual int currPos() const = 0;
53
54      // Set current position
55      virtual void moveToPos(int pos) = 0;
56
57      // Return the current element
58      virtual const E& getValue() const = 0;
59  };
60
61  template <typename E>  // Singly linked list node
62  class Link
63  {
64  public:
65      E element;   // Value for this node
66      Link* next;  // Pointer to next node in list
67
68      Link(const E& elemval, Link* nextval = NULL)
69      {
70          element = elemval;
71          next = nextval;
72      }
73      Link(Link* nextval = NULL) { next = nextval; }
74  };
75
76  template <typename E>  // Linked list implementation
77  class LList : public List<E>
78  {
79  private:
80      Link<E>* head;  // Pointer to list header
81      Link<E>* tail;  // Pointer to last element
82      Link<E>* curr;  // Access to current element
83      int cnt;        // Size of list
```

```
84
85      void init()  // Initialization helper method
86      {
87          curr = tail = head = new Link<E>;
88          cnt = 0;
89      }
90
91      void removeall()  // Return link node to free store
92      {
93          while (head != NULL) {
94              curr = head;
95              head = head->next;
96              delete curr;
97          }
98      }
99
100 public:
101     LList(int size = 5) { init(); }  // Constructor
102     ~LList() { removeall(); }        // Destructor
103
104     void print() const;
105     void clear()
106     {
107         removeall();
108         init();
109     }
110
111     // Insert "it" at current position
112     void insert(const E& it)
113     {
114         curr->next = new Link<E>(it, curr->next);
115         if (tail == curr)
116             tail = curr->next;  // New tail
117     }
118
119     void append(const E& it)
120     {
121         tail = tail->next = new Link<E>(it, NULL);
122         cnt++;
123     }
124
125     // Remove and return current element
126     E remove()
127     {
128         Assert(curr->next != NULL, "No element");
129         E it = curr->next->element;   // Remember value
130         Link<E>* ltemp = curr->next;  // Remember link node
131         if (tail == curr->next)
132             tail = curr;  // Reset tail
133         curr->next = curr->next->next;
134         delete ltemp;
135         cnt--;
136         return it;
137     }
138
139     void moveToStart() { curr = head; }
140     void moveToEnd() { curr = tail; }
141
```

```
142     void prev()
143     {
144         if (curr == head)
145             return;
146         Link<E>* temp = head;
147         while (temp->next != curr)
148             temp = temp->next;
149         curr = temp;
150     }
151 
152     void next()
153     {
154         if (curr != tail)
155             curr = curr->next;
156     }
157 
158     int length() const { return cnt; }
159 
160     int currPos() const
161     {
162         Link<E>* temp = head;
163         int i;
164         for (i = 0; curr != temp; i++)
165             temp = temp->next;
166         return i;
167     }
168 
169     void moveToPos(int pos)
170     {
171         Assert((pos >= 0) && (pos <= cnt), "Position out of range");
172         curr = head;
173         for (int i = 0; i < pos; i++)
174             curr = curr->next;
175     }
176 
177     const E& getValue() const
178     {
179         Assert(curr->next != NULL, "No value");
180         return curr->next->element;
181     }
182 };
```

Author: XF

Finished time: 2019/11/3