4.1

List.moveToPos(2);

List.remove();

4.5

(a):curr在末尾时,curr和tail指向同一节点,且该节点next为NULL.替代方法中插入时

需要利用curr->next新建一个节点,此时会报出空指针错误.应该提前判断curr和tail

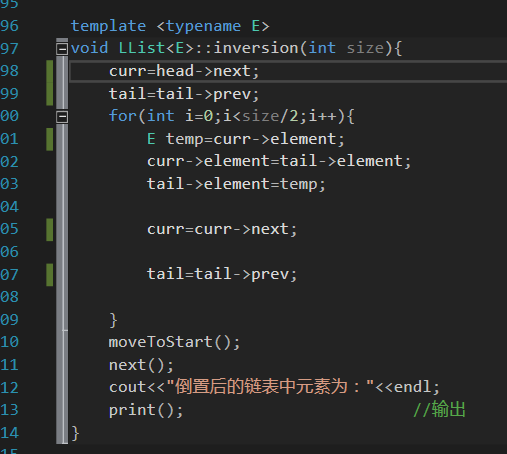
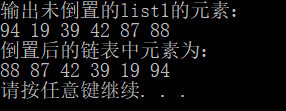
是否指向同一节点,如果是,则将替代方法中的insert换成append.相比4.1.2的代码,这

种方法并不简洁.

(b):curr指向当前结点的情况下,删除操作需要调用prev来获取前驱节点,其复杂度为

θ(n).可以连续删除.

4.6

template <typename E>

void LList<E>::reverse(int cnt)

{

curr=head->next;

tail=tail->prev;

for(int i=0;i<cnt/2;i++)

{

E temp=curr->element;

curr->element=tail->element;

tail->element=temp;

curr=curr->next;

tail=tail->prev;

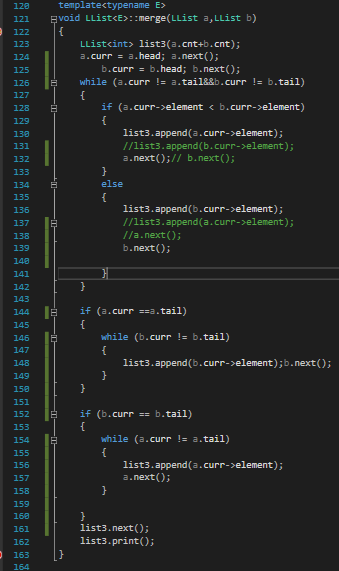
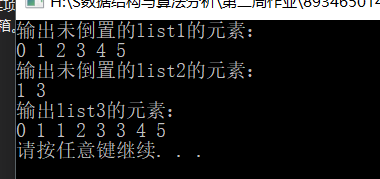
}

moveToStart();

next();

}

4.7

template<typename E>

void LList<E>::merge(LList a,LList b)

{

LList<int> list3(a.cnt+b.cnt);

a.curr = a.head; a.next();

b.curr = b.head; b.next();

while (a.curr != a.tail&&b.curr != b.tail)

{

if (a.curr->element < b.curr->element)

{

list3.append(a.curr->element);

a.next();

}

else

{

list3.append(b.curr->element);

b.next();

}

}

if (a.curr ==a.tail)

{

while (b.curr != b.tail)

{

list3.append(b.curr->element);b.next();

}

}

if (b.curr == b.tail)

{

while (a.curr != a.tail)

{

list3.append(a.curr->element);

a.next();

}

}

list3.next();

list3.print();

}

4.8

(a): template <class Elem>

void LList<Elem>::init(const int size)

{

head = tail = curr = new link;

head->next = head;

}

template <class Elem>

void LList<Elem>::clear()

{

while (head->next != NULL)

{

curr = head->next;

head->next = curr->next;

delete curr;

}

tail = curr = head->next = head;

}

template <class Elem>

void LList<Elem>::insert(const Elem& item)

{

assert(curr != NULL);

curr->next = new link(item, curr->next);

if (tail->next != head) tail = tail->next;

}

template <class Elem>

void LList<Elem>::append(const Elem& item)

{ tail = tail->next = new link(item, head); }

template <class Elem>

void LList<Elem>::next()

{ curr = curr->next; }

template <class Elem>

void LList<Elem>::prev()

{

link\* temp = curr;

while (temp->next!=curr) temp=temp->next;

curr = temp;

}

(b): template <class Elem>

void LList<Elem>::init(const int size)

{

head = tail = curr = new link;

head->next = head;

head->prev = head;

}

template <class Elem>

Elem LList<Elem>::remove()

{

if (curr->next==tail)

return NULL;

Elem it =curr->next->element;

Link<E>\* ltemp=curr->next;

cur->next->next->prev=curr;

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

delete ltemp;

return it;

}

template <class Elem>

void LList<Elem>::insert(const Elem& item)

{

assert(curr != NULL);

curr->next =curr->next->prev= new link(item, curr, curr->next);

if (tail->next != head) tail = tail->next;

}

template <class Elem>

void LList<Elem>::append(const Elem& item)

{ tail->prev = tail->prev->next = new link(item,tail->prev, tail); }

template <class Elem>

void LList<Elem>::next()

{ curr = curr->next; }

template <class Elem>

void LList<Elem>::prev()

{

if(curr!=head)

curr=curr->prev;

else

curr=tail;

}

4.12

(a):设E为数据元素的存储单元大小,D为数组中存储的线性表元素的最大数目,P为指

针存储单元大小,则P=4,E=4,由公式:

n>DE/(P+E) 得:

n>4D/8=D/2,所以当n小于D/2时,空间效率上顺序表更高效

(b): 设E为数据元素的存储单元大小,D为数组中存储的线性表元素的最大数目,P为指

针存储单元大小,则P=4,E=8,由公式:

n>DE/(P+E) 得:

n>4D/12=2D/3,所以当n小于2D/3时,空间效率上顺序表更高效