（一）

1.循环队列的改造

改造后的代码如下

主要思路是通过length来表示原来rear所指代的内容

template<class ElemType>

class SeqQueue

{

protected:

int front, length; // 队头队尾指针

int maxSize; // 队列容量

ElemType \*elems; // 元素存储空间

public:

SeqQueue(int size = DEFAULT\_SIZE); // 构造函数

virtual ~SeqQueue(); // 析构函数

int GetLength() const; // 求队列长度

bool IsEmpty() const; // 判断队列是否为空

void Clear(); // 将队列清空

void Traverse(void (\*Visit)(const ElemType &)) const;// 遍历队列

Status DelQueue(ElemType &e); // 出队操作

Status GetHead(ElemType &e) const; // 取队头操作

Status EnQueue(const ElemType e); // 入队操作

SeqQueue(const SeqQueue<ElemType> &q); // 复制构造函数

SeqQueue<ElemType> &operator =(const SeqQueue<ElemType> &q);// 赋值语句重载

};

// 循环队列类的实现部分

template<class ElemType>

SeqQueue<ElemType>::SeqQueue(int size)

// 操作结果：构造一个容量为size的空循环队列

{

maxSize = size; // 设置队列容量

elems = new ElemType[maxSize]; // 分配元素存储空间

length = front = 0; // 初始化队头与队尾

}

template <class ElemType>

SeqQueue<ElemType>::~SeqQueue()

// 操作结果：销毁队列

{

delete []elems; // 释放元素存储空间

}

template<class ElemType>

int SeqQueue<ElemType>::GetLength() const

// 操作结果：返回队列长度

{

return length;

}

template<class ElemType>

bool SeqQueue<ElemType>::IsEmpty() const

// 操作结果：如队列为空，则返回true，否则返回false

{

return length==0;

}

template<class ElemType>

void SeqQueue<ElemType>::Clear()

// 操作结果：清空队列

{

length = front = 0;

}

template <class ElemType>

void SeqQueue<ElemType>::Traverse(void (\*Visit)(const ElemType &)) const

// 操作结果：依次对队列的每个元素调用函数(\*visit)

{

int now=front;

for (int i = 1; i <=length ; i++)

{

(\*Visit)(elems[now]);

now=(now+1)%maxSize;

}

}

template<class ElemType>

Status SeqQueue<ElemType>::DelQueue(ElemType &e)

// 操作结果：如果队列非空，那么删除队头元素，并用e返回其值，函数返回SUCCESS,

// 否则函数返回UNDER\_FLOW，

{

if (!IsEmpty()) { // 队列非空

e = elems[front]; // 用e返回队头元素

front = (front + 1) % maxSize; // front指向下一元素

length--;

return SUCCESS;

}

else // 队列为空

return UNDER\_FLOW;

}

template<class ElemType>

Status SeqQueue<ElemType>::GetHead(ElemType &e) const

// 操作结果：如果队列非空，那么用e返回队头元素，函数返回SUCCESS,

// 否则函数返回UNDER\_FLOW，

{

if (!IsEmpty()) { // 队列非空

e = elems[front]; // 用e返回队头元素

return SUCCESS;

}

else

return UNDER\_FLOW;

}

template<class ElemType>

Status SeqQueue<ElemType>::EnQueue(const ElemType e)

// 操作结果：如果队列已满，返回OVER\_FLOW,

// 否则插入元素e为新的队尾，返回SUCCESS

{

if (length == maxSize)

return OVER\_FLOW;

else { // 队列未满，入队成功

elems[(front+length)%maxSize] = e;

length++; // 插入e为新队尾

return SUCCESS;

}

}

template<class ElemType>

SeqQueue<ElemType>::SeqQueue(const SeqQueue<ElemType> &q)

// 操作结果：由队列q构造新队列--复制构造函数

{

maxSize = q.maxSize; // 设置队列容量

if (elems != NULL) delete []elems; // 释放存储空间

elems = new ElemType[maxSize]; // 分配存储空间

front = q.front; // 复制队头位置

length = q.length;

int now=front; // 复制队尾位置

for (int i = 0; i <=length; i ++)

{

elems[now]=q.elems[now];

now=(now+1)%maxSize;

}

}

template<class ElemType>

SeqQueue<ElemType> &SeqQueue<ElemType>::operator =(const SeqQueue<ElemType> &q)

// 操作结果：将队列q赋值给当前队列--赋值语句重载

{

if (&q != this) {

maxSize = q.maxSize; // 设置队列容量

if (elems != NULL) delete []elems; // 释放存储空间

elems = new ElemType[maxSize]; // 分配存储空间

front = q.front; // 复制队头位置

length = q.length;

int now=front; // 复制队尾位置

for (int i = 0; i <=length; i ++)

{

elems[now]=q.elems[now];

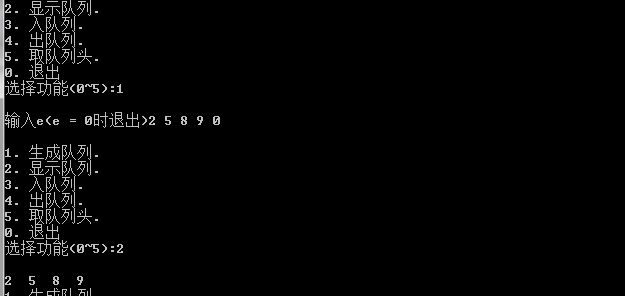
now=(now+1)%maxSize;

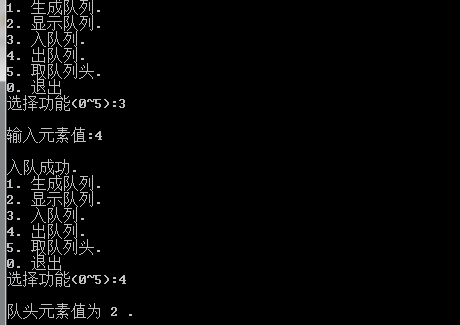
}

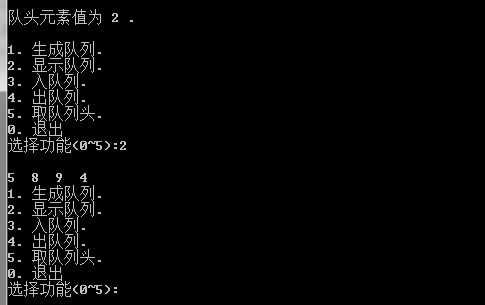
}

return \*this;

}







2.链队列的改造

主要思路是用rear->next来代替原来的front指针

template<class ElemType>

class LinkQueue

{

protected:

// 链队列实现的数据成员:

Node<ElemType> \*rear; // 队头队尾指指

public:

LinkQueue(); // 无参数的构造函数

virtual ~LinkQueue(); // 析构函数

int GetLength() const; // 求队列长度

bool IsEmpty() const; // 判断队列是否为空

void Clear(); // 将队列清空

void Traverse(void (\*Visit)(const ElemType &)) const ; // 遍历队列

Status DelQueue(ElemType &e); // 出队操作

Status GetHead(ElemType &e) const; // 取队头操作

Status EnQueue(const ElemType e); // 入队操作

LinkQueue(const LinkQueue<ElemType> &q); // 复制构造函数

LinkQueue<ElemType> &operator =(const LinkQueue<ElemType> &q);// 赋值语句重载

};

// 链队列类的实现部分

template<class ElemType>

LinkQueue<ElemType>::LinkQueue()

// 操作结果：构造一个空队列

{

rear = new Node<ElemType>; // 生成链队列头结点

}

template<class ElemType>

LinkQueue<ElemType>::~LinkQueue()

// 操作结果：销毁队列

{

Clear();

}

template<class ElemType>

int LinkQueue<ElemType>::GetLength() const

// 操作结果：返回队列长度

{

int count = 0; // 初始化计数器

for (Node<ElemType> \*p = rear->next; p == rear; p = p->next)

count++; // 统计链队列中的结点数

return count;

}

template<class ElemType>

bool LinkQueue<ElemType>::IsEmpty() const

// 操作结果：如队列为空，则返回true，否则返回false

{

return rear == NULL;

}

template<class ElemType>

void LinkQueue<ElemType>::Clear()

// 操作结果：清空队列

{

Node<ElemType> \*p = rear->next;

while (p != NULL) { // 依次删除队列中的元素结点

rear->next = p->next;

delete p;

if(p!=NULL)p = rear->next;

}

rear = NULL;

}

template <class ElemType>

void LinkQueue<ElemType>::Traverse(void (\*Visit)(const ElemType &)) const

// 操作结果：依次对队列的每个元素调用函数(\*visit)

{

for (Node<ElemType> \*p = rear->next; p == rear; p = p->next)

// 对队列每个元素调用函数(\*visit)访问

(\*Visit)(p->data);

}

template<class ElemType>

Status LinkQueue<ElemType>::DelQueue(ElemType &e)

// 操作结果：如果队列非空，那么删除队头元素，并用e返回其值，函数返回SUCCESS,

// 否则函数返回UNDER\_FLOW，

{

if (!IsEmpty()) { // 队列非空

Node<ElemType> \*p = rear->next; // 指向队列头素

e = p->data; // 用e返回队头元素

if (rear == p) delete rear;

else {rear->next = p->next; // front指向下一元素

delete p;} // 释放出队的元素结点

return SUCCESS;

}

else

return UNDER\_FLOW;

}

template<class ElemType>

Status LinkQueue<ElemType>::GetHead(ElemType &e) const

// 操作结果：如果队列非空，那么用e返回队头元素，函数返回SUCCESS,

// 否则函数返回UNDER\_FLOW，

{

if (!IsEmpty()) { // 队列非空

e = rear->next->data; // 用e返回队头元素

return SUCCESS;

}

else

return UNDER\_FLOW;

}

template<class ElemType>

Status LinkQueue<ElemType>::EnQueue(const ElemType e)

// 操作结果：如果系统空间不够，返回OVER\_FLOW,

// 否则插入元素e为新的队尾，返回SUCCESS

{

Node<ElemType> \*p;

p = new Node<ElemType>(e);

Node<ElemType> \*front; // 生成一个新结点

if(rear==NULL){rear=rear->next=p;}

else

{

if (p)

{

front= rear->next;

rear->next = p; // 将新结点加在队尾

rear = rear->next;

rear->next= front; // rear指向新队尾

return SUCCESS;

}

else //系统空间不够，返回OVER\_FLOW

return OVER\_FLOW;

}

}

template<class ElemType>

LinkQueue<ElemType>::LinkQueue(const LinkQueue<ElemType> &q)

// 操作结果：由队列q构造新队列--复制构造函数

{

rear = NULL;// 生成队列头结点

for (Node<ElemType> \*p = q.rear->next; p == rear; p = p->next)

// 取q队列每个元素的值,将其在当前队列中作入队列操作

EnQueue(p->data);

}

template<class ElemType>

LinkQueue<ElemType> &LinkQueue<ElemType>::operator =(const LinkQueue<ElemType> &q)

// 操作结果：将队列q赋值给当前队列--赋值语句重载

{

if (&q != this) {

Clear(); //清除原有队列

for (Node<ElemType> \*p = q.rear->next; p ==rear; p = p->next)

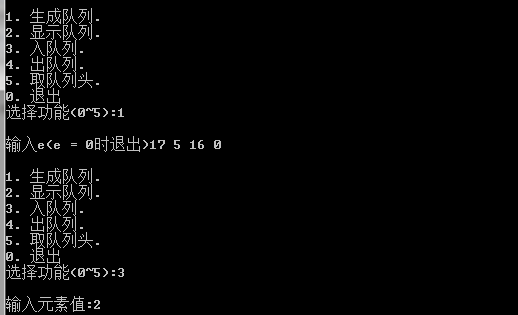
// 取q队列每个元素的值,将其在当前队列中作入队列操作

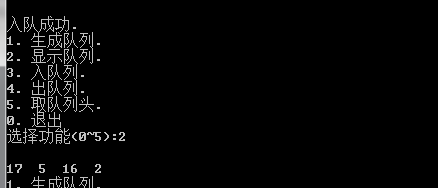
EnQueue(p->data);

}

return \*this;

}





(二)

3.双端队列的类模板

template<class ElemType>

class LinkQueue

{

protected:

// 链队列实现的数据成员:

Node<ElemType> \*front,\*rear; // 队头队尾指指

public:

LinkQueue(); // 无参数的构造函数

virtual ~LinkQueue(); // 析构函数

int GetLength() const; // 求队列长度

bool IsEmpty() const; // 判断队列是否为空

void Clear(); // 将队列清空

void Traverse(void (\*Visit)(const ElemType &)) const ; // 遍历队列

Status DelQueue(int k,ElemType &e); // 出队操作

Status GetHead(ElemType &e) const; // 取队头操作

Status EnQueue(int k,const ElemType e); // 入队操作

LinkQueue(const LinkQueue<ElemType> &q); // 复制构造函数

LinkQueue<ElemType> &operator =(const LinkQueue<ElemType> &q);// 赋值语句重载

};

// 链队列类的实现部分

template<class ElemType>

LinkQueue<ElemType>::LinkQueue()

// 操作结果：构造一个空队列

{

rear = front = NULL; // 生成链队列头结点

}

template<class ElemType>

LinkQueue<ElemType>::~LinkQueue()

// 操作结果：销毁队列

{

Clear();

}

template<class ElemType>

int LinkQueue<ElemType>::GetLength() const

// 操作结果：返回队列长度

{

int count = 0; // 初始化计数器

for (Node<ElemType> \*p = front; p != NULL; p = p->next)

count++; // 统计链队列中的结点数

return count;

}

template<class ElemType>

bool LinkQueue<ElemType>::IsEmpty() const

// 操作结果：如队列为空，则返回true，否则返回false

{

return rear == front;

}

template<class ElemType>

void LinkQueue<ElemType>::Clear()

// 操作结果：清空队列

{

Node<ElemType> \*p = front;

while (p != NULL) { // 依次删除队列中的元素结点

front->next = p->next;

delete p;

p = front->next;

}

rear = front;

}

template <class ElemType>

void LinkQueue<ElemType>::Traverse(void (\*Visit)(const ElemType &)) const

// 操作结果：依次对队列的每个元素调用函数(\*visit)

{

for (Node<ElemType> \*p = front; p != NULL; p = p->next)

// 对队列每个元素调用函数(\*visit)访问

(\*Visit)(p->data);

}

template<class ElemType>

Status LinkQueue<ElemType>::DelQueue(int k,ElemType &e)

// 操作结果：如果队列非空，那么删除队头元素，并用e返回其值，函数返回SUCCESS,

// 否则函数返回UNDER\_FLOW，

{

if (!IsEmpty())

{ // 队列非空

if(front==rear)

{

delete front;

front=rear=NULL;

}

else

{

if (k==0)

{

Node<ElemType> \*p = front ;// 指向队列头素

e = p->data; // 用e返回队头元素

front = p->next;

}

if(k==1)

{

Node<ElemType> \*p = rear;

e=p->data;

delete rear;

for (Node<ElemType> \*p = front; p != NULL; p = p->next)

{

rear=p;

}

}

}

return SUCCESS;

}

else

return UNDER\_FLOW;

}

template<class ElemType>

Status LinkQueue<ElemType>::GetHead(ElemType &e) const

// 操作结果：如果队列非空，那么用e返回队头元素，函数返回SUCCESS,

// 否则函数返回UNDER\_FLOW，

{

if (!IsEmpty()) { // 队列非空

e = front->data; // 用e返回队头元素

return SUCCESS;

}

else

return UNDER\_FLOW;

}

template<class ElemType>

Status LinkQueue<ElemType>::EnQueue(int k,const ElemType e)

// 操作结果：如果系统空间不够，返回OVER\_FLOW,

// 否则插入元素e为新的队尾，返回SUCCESS

{

Node<ElemType> \*p;

p = new Node<ElemType>(e); // 生成一个新结点

if (front==NULL)

{

front=rear=p;

}

else

{

if(k==1)

{

rear->next = p; // 将新结点加在队尾

rear = rear->next; // rear指向新队尾

return SUCCESS;

}

if(k==0)

{

Node<ElemType> \*q;

q=front;

front=p;

front->next=q;

}

}

}

template<class ElemType>

LinkQueue<ElemType>::LinkQueue(const LinkQueue<ElemType> &q)

// 操作结果：由队列q构造新队列--复制构造函数

{

rear = front = new Node<ElemType>; // 生成队列头结点

for (Node<ElemType> \*p = q.front; p != NULL; p = p->next)

// 取q队列每个元素的值,将其在当前队列中作入队列操作

EnQueue(p->data);

}

template<class ElemType>

LinkQueue<ElemType> &LinkQueue<ElemType>::operator =(const LinkQueue<ElemType> &q)

// 操作结果：将队列q赋值给当前队列--赋值语句重载

{

if (&q != this) {

Clear(); //清除原有队列

for (Node<ElemType> \*p = q.front; p != NULL; p = p->next)

// 取q队列每个元素的值,将其在当前队列中作入队列操作

EnQueue(p->data);

}

return \*this;

}