```c
/* OrderVector.c */
#include <stdlib.h>
#include <malloc.h>
#include <string.h>
#include <assert.h>
#include <limits.h>
#include "OrderVector.h"

//线性表数据堆空间倍增
static void VectorGrow(VECTOR *v)
{
    v->capacity *= 2;
    v->elems = realloc(v->elems, v->elemSize * v->capacity);
    assert(NULL != v->elems);
}

//线性表数据堆空间倍减
static void VectorReduce(VECTOR *v)
{
    if (v->size <= INITALLOC)
    {
        return;
    }
    v->capacity /= 2;
    v->elems = realloc(v->elems, v->elemSize * v->capacity);
    assert(NULL != v->elems);
}

//新建线性表
void VectorNew(VECTOR *v, int elemSize, int capacity, int fSupportGrow, VectorCmp
*cmpFn, VectorFree *freeFn)
{
    assert(elemSize > 0);
    assert(capacity > 0);
    assert((fSupportGrow == 0) || (fSupportGrow == 1));
    v->elemSize = elemSize;
    v->size = 0;
    v->capacity = capacity;
    v->fSupportGrow = fSupportGrow;
    v->elems = malloc(elemSize * v->capacity);
    assert(NULL != v->elems);
    v->cmpFn = cmpFn;
    v->freeFn = freeFn;
}

//销毁线性表
void VectorDispose(VECTOR *v)
{
    if (NULL != v->freeFn)
    {
        int i;
        for(i = 0; i < v->size; i++)
        {
            v->freeFn((char *)v->elems + i * v->elemSize);
        }
    }
    free(v->elems);
    v->elems = NULL;
    v->size = 0;
}

//判断线性表是否已满
int VectorFull(VECTOR *v)
{
    return (v->size == v->capacity);
}

//判断线性表是否为空
int VectorEmpty(VECTOR *v)
{
    return (0 == v->size);
}

```

```
73  //线性表元素数量
74  int VectorSize(VECTOR *v)
75  {
76      return v->size;
77  }
78
79  //清空线性表元素
80  void VectorMakeEmpty(VECTOR *v)
81  {
82      if (NULL != v->freeFn)
83      {
84          int i;
85          for(i = 0; i < v->size; i++)
86          {
87              v->freeFn((char *)v->elems + i * v->elemSize);
88          }
89      }
90      v->size = 0;
91      v->elems = realloc(v->elems, v->elemSize * v->capacity);
92      assert(NULL != v->elems);
93  }
94
95  //根据位置查找元素，返回值为元素地址
96  void *VectorGetByPos(VECTOR *v, int pos)
97  {
98      if ((pos < 0) || (pos >= v->size))
99      {
100         return NULL;
101     }
102     return (char *)v->elems + pos * v->elemSize;
103 }
104
105 //线性查找算法
106 static int linearSearch(VECTOR *v, const void *e)
107 {
108     int pos = 0;
109     for (; pos < v->size; pos ++)
110     {
111         if (v->cmpFn(e, (char *)v->elems + pos * v->elemSize) <= 0)
112         {
113             break;
114         }
115     }
116     return pos;
117 }
118
119 //在有序向量的区间[lo, hi)内二分查找e
120 static int binSearch(VECTOR *v, const void *e, int lo, int hi)
121 {
122     while (lo < hi) //成功查找不能提前终止循环
123     {
124         int mi = (lo + hi) >> 1;
125         if (0 >= v->cmpFn(e, VectorGetByPos(v, mi)))
126         {
127             hi = mi;
128         }
129         else
130         {
131             lo = mi + 1;
132         }
133     }
134     return lo;
135 }
136
137 //根据值查找元素，way!=0线性查找，way=0二分查找，返回值为不小于该元素的最小位置
138 int VectorSearch(VECTOR *v, const void *e, int way)
139 {
140     return ((0 != way) ? linearSearch(v, e) : binSearch(v, e, 0, VectorSize(v)));
141 }
142
143 //判断关键码是否在向量的第pos个置位，返回值：0--不在，!0--存在
144 int VectorFind(VECTOR *v, int pos, const void *e)
145 {
```

```
146         if ((pos < 0) || (pos >= VectorSize(v)) || (NULL == v->cmpFn))
147         {
148             return 0;
149         }
150         return (0 == v->cmpFn((char *)v->elems + pos * v->elemSize, e));
151     }
152 
153     //根据位置插入元素（慎用，可能会破坏有序性），返回值：!0--插入失败，0--插入成功
154     int VectorInsertByPos(VECTOR *v, const void *e, int pos)
155     {
156         if ((pos > v->size) || pos < 0)
157         {
158             return -1;
159         }
160         if (VectorFull(v) && (!v->fSupportGrow))
161         {
162             return -1;
163         }
164         else if (VectorFull(v) && v->fSupportGrow)
165         {
166             VectorGrow(v);
167         }
168         void *target = (char *)v->elems + v->elemSize * pos;
169         if (pos != v->size) //如果不是在末尾插入就需要移动一部分元素
170         {
171             memmove((char *)target + v->elemSize, target, v->elemSize * (v->size - pos));
172         }
173         memcpy(target, e, v->elemSize);
174         v->size ++;
175         return 0;
176     }
177 
178     //插入元素，返回值：!0--插入失败，0--插入成功
179     int VectorInsert(VECTOR *v, const void *e)
180     {
181         //有序表已满且不允许扩容时插入失败
182         if (VectorFull(v) && (!v->fSupportGrow))
183         {
184             return -1;
185         }
186         else if (VectorFull(v) && v->fSupportGrow)
187         {
188             VectorGrow(v);
189         }
190         int pos = VectorSearch(v, e, 0);
191         //待插入的元素已经存在，插入失败
192         if ((pos < v->size) && VectorFind(v, pos, e))
193         {
194             return -1;
195         }
196         return VectorInsertByPos(v, e, pos);
197     }
198 
199     //根据位置删除元素，返回值：!0--删除失败，0--删除成功
200     int VectorRemoveByPos(VECTOR *v, int pos)
201     {
202         if ((pos < 0) || (pos >= v->size) || VectorEmpty(v))
203         {
204             return -1;
205         }
206         void *target = (char *)v->elems + v->elemSize * pos;
207         if (NULL != v->freeFn)
208         {
209             v->freeFn(target);
210         }
211         if (pos != (v->size - 1))
212         {
213             memmove(target, (char *)target + v->elemSize, v->elemSize * (v->size - pos -
                1));
214         }
215         v->size --;
216         if ((v->size * 2 <= v->capacity) && v->fSupportGrow)
217         {
```

```
218             VectorReduce(v);
219         }
220         return 0;
221     }
222
223     //删除元素，返回值：!0--删除失败，0--删除成功
224     int VectorRemove(VECTOR *v, void *e)
225     {
226         if (VectorEmpty(v))
227         {
228             return -1;
229         }
230         int pos = VectorSearch(v, e, 0);
231         //待删除的元素不存在，删除失败
232         if ((pos >= v->size) || !VectorFind(v, pos, e))
233         {
234             return -1;
235         }
236         return VectorRemoveByPos(v, pos);
237     }
238
239     //根据位置删除元素（无需深度删除），返回值：!0--删除失败，0--删除成功
240     int VectorRemoveByPosU(VECTOR *v, int pos)
241     {
242         if ((pos < 0) || (pos >= v->size) || VectorEmpty(v))
243         {
244             return -1;
245         }
246         void *target = (char *)v->elems + v->elemSize * pos;
247         if (pos != (v->size - 1))
248         {
249             memmove(target, (char *)target + v->elemSize, v->elemSize * (v->size - pos -
                1));
250         }
251         v->size --;
252         if ((v->size * 2 <= v->capacity) && v->fSupportGrow)
253         {
254             VectorReduce(v);
255         }
256         return 0;
257     }
258
259     //删除元素（无需深度删除），返回值：!0--删除失败，0--删除成功
260     int VectorRemoveU(VECTOR *v, void *e)
261     {
262         if (VectorEmpty(v))
263         {
264             return -1;
265         }
266         int pos = VectorSearch(v, e, 0);
267         //待删除的元素不存在，删除失败
268         if ((pos >= v->size) || !VectorFind(v, pos, e))
269         {
270             return -1;
271         }
272         return VectorRemoveByPosU(v, pos);
273     }
274
275     //遍历线性表
276     void VectorTraverse(VECTOR *v, VectorTraverseOp *traverseOpFn, void *outData)
277     {
278         if (NULL == traverseOpFn)
279         {
280             return ;
281         }
282         void *elemAddr;
283         int i = 0;
284         for (; i < v->size; i ++)
285         {
286             elemAddr = (char *)v->elems + i * v->elemSize;
287             traverseOpFn(elemAddr, outData);
288         }
289     }
```

```c

//交换两个表的元素，返回值：!0--交换失败，0--交换成功
int VectorSwap(VECTOR *v, VECTOR *u, int rankV, int rankU)
{
    if (v->elemSize != u->elemSize)
    {
        return -1;
    }
    if (rankV < 0 || rankV >= v->size || rankU < 0 || rankU >= u->size)
    {
        return -1;
    }
    int size = v->elemSize;
    void *tmp = malloc(size);
    if (NULL == tmp)
    {
        return -1;
    }
    memcpy(tmp, (char *)v->elems + rankV * size, size);
    memcpy((char *)v->elems + rankV * size, (char *)u->elems + rankU * size, size);
    memcpy((char *)u->elems + rankU * size, tmp, size);
    free(tmp);
    return 0;
}
```