```c
/* Vector.c */
#include <stdlib.h>
#include <malloc.h>
#include <string.h>
#include <assert.h>
#include "Vector.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;
}

//初始化线性表内容
void VectorInit(VECTOR *v, int c)
{
    if (0 != v->size)
    {
        return ;
    }
    memset(v->elems, c, v->elemSize * v->capacity);
}

//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)
{
```

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

```
146 //根据值查找元素，way!=0线性查找，way=0二分查找
147 int VectorSearch(VECTOR *v, const void *e, int way)
148 {
149     return ((0 != way) ? linearSearch(v, e) : binSearch(v, e, 0, VectorSize(v)));
150 }
151
152 //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
153 int VectorFind(VECTOR *v, int pos, const void *e)
154 {
155     if ((pos < 0) || (pos >= VectorSize(v)) || (NULL == v->cmpFn))
156     {
157         return 0;
158     }
159     return (0 == v->cmpFn((char *)v->elems + pos * v->elemSize, e));
160 }
161
162 //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
163 int VectorInsertByPos(VECTOR *v, const void *e, int pos)
164 {
165     if ((pos > v->size) || pos < 0)
166     {
167         return -1;
168     }
169     if (VectorFull(v) && (!v->fSupportGrow))
170     {
171         return -1;
172     }
173     else if (VectorFull(v) && v->fSupportGrow)
174     {
175         VectorGrow(v);
176     }
177     void *target = (char *)v->elems + v->elemSize * pos;
178     if (pos != v->size) //如果不是在末尾插入就需要移动一部分元素
179     {
180         memmove((char *)target + v->elemSize, target, v->elemSize * (v->size - pos));
181     }
182     memcpy(target, e, v->elemSize);
183     v->size ++;
184     return 0;
185 }
186
187 //插入元素，返回值：!0--插入失败，0--插入成功
188 int VectorInsert(VECTOR *v, const void *e)
189 {
190     //有序表已满且不允许扩容时插入失败
191     if (VectorFull(v) && (!v->fSupportGrow))
192     {
193         return -1;
194     }
195     else if (VectorFull(v) && v->fSupportGrow)
196     {
197         VectorGrow(v);
198     }
199     //待插入的元素已经存在，插入失败
200     if (VectorSearch(v, e, !0) >= 0)
201     {
202         return -1;
203     }
204     return VectorInsertByPos(v, e, VectorSize(v));
205 }
206
207 //根据位置删除元素，返回值：!0--删除失败，0--删除成功
208 int VectorRemoveByPos(VECTOR *v, int pos)
209 {
210     if ((pos < 0) || (pos >= v->size) || VectorEmpty(v))
211     {
212         return -1;
213     }
214     void *target = (char *)v->elems + v->elemSize * pos;
215     if (NULL != v->freeFn)
216     {
217         v->freeFn(target);
218     }
```

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

```c
void VectorTraverse(VECTOR *v, VectorTraverseOp *traverseOpFn, void *outData)
{
    if (NULL == traverseOpFn)
    {
        return ;
    }
    void *elemAddr;
    int i = 0;
    for (; i < v->size; i ++)
    {
        elemAddr = (char *)v->elems + i * v->elemSize;
        traverseOpFn(elemAddr, outData);
    }
}

//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;
}

//线性表排序，mode：0--顺序，!0--逆序
void ListSort(VECTOR *l, int mode)
{
    return ;
}
```