```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)
{
    return (v->size == v->capacity);
}

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

//线性表元素数量
int VectorSize(VECTOR *v)
{
    return v->size;
}

//清空线性表元素
void VectorMakeEmpty(VECTOR *v)
{
    if (NULL!= v->freeFn)
    {
        int i;
        for(i = 0; i < v->size; i++)
        {
            v->freeFn((char *)v->elems + i * v->elemSize);
        }
    }
    v->size = 0;
    v->elems = realloc(v->elems, v->elemSize * v->capacity);
    assert(NULL != v->elems);
}

//根据位置查找元素，返回值为元素地址
void *VectorGetByPos(VECTOR *v, int pos)
{
    if ((pos < 0) || (pos >= v->size))
    {
        return NULL;
    }
    return (char *)v->elems + pos * v->elemSize;
}

//线性查找算法，返回值：<0--没找到，>=0--元素所在地址
static int linearSearch(VECTOR *v, const void *e)
{
    int pos = 0;
    for (; pos < v->size; pos ++)
    {
        if (v->cmpFn(e, (char *)v->elems + pos * v->elemSize) == 0)
        {
            return pos;
        }
    }
    return -1;
}

//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
static int binSearch(VECTOR *v, const void *e, int lo, int hi)
{
    while (lo < hi) //成功查找不能提前终止循环
    {
        int mi = (lo + hi) >> 1;
        if (0 >= v->cmpFn(e, VectorGetByPos(v, mi)))
        {
            hi = mi;
        }
        else
        {
            lo = mi + 1;
        }
    }
    return lo;
}
```

```c
//根据值查找元素，way!=0线性查找，way=0二分查找
int VectorSearch(VECTOR *v, const void *e, int way)
{
    return ((0 != way) ? linearSearch(v, e) : binSearch(v, e, 0, VectorSize(v)));
}

//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
int VectorFind(VECTOR *v, int pos, const void *e)
{
    if ((pos < 0) || (pos >= VectorSize(v)) || (NULL == v->cmpFn))
    {
        return 0;
    }
    return (0 == v->cmpFn((char *)v->elems + pos * v->elemSize, e));
}

//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
int VectorInsertByPos(VECTOR *v, const void *e, int pos)
{
    if ((pos > v->size) || pos < 0)
    {
        return -1;
    }
    if (VectorFull(v) && (!v->fSupportGrow))
    {
        return -1;
    }
    else if (VectorFull(v) && v->fSupportGrow)
    {
        VectorGrow(v);
    }
    void *target = (char *)v->elems + v->elemSize * pos;
    if (pos != v->size) //如果不是在末尾插入就需要移动一部分元素
    {
        memmove((char *)target + v->elemSize, target, v->elemSize * (v->size - pos));
    }
    memcpy(target, e, v->elemSize);
    v->size ++;
    return 0;
}

//插入元素，返回值：!0--插入失败，0--插入成功
int VectorInsert(VECTOR *v, const void *e)
{
    //有序表已满且不允许扩容时插入失败
    if (VectorFull(v) && (!v->fSupportGrow))
    {
        return -1;
    }
    else if (VectorFull(v) && v->fSupportGrow)
    {
        VectorGrow(v);
    }
    //待插入的元素已经存在，插入失败
    if (VectorSearch(v, e, !0) >= 0)
    {
        return -1;
    }
    return VectorInsertByPos(v, e, VectorSize(v));
}

//根据位置删除元素，返回值：!0--删除失败，0--删除成功
int VectorRemoveByPos(VECTOR *v, int pos)
{
    if ((pos < 0) || (pos >= v->size) || VectorEmpty(v))
    {
        return -1;
    }
    void *target = (char *)v->elems + v->elemSize * pos;
    if (NULL != v->freeFn)
    {
        v->freeFn(target);
    }
    if (pos != (v->size - 1))
    {
        memmove(target, (char *)target + v->elemSize, v->elemSize * (v->size - pos - 1));
    }
    v->size --;
    if ((v->size * 2 <= v->capacity) && v->fSupportGrow)
    {
        VectorReduce(v);
    }
    return 0;
}

//删除元素，返回值：!0--删除失败，0--删除成功
int VectorRemove(VECTOR *v, void *e)
{
    if (VectorEmpty(v))
    {
        return -1;
    }
    int pos = VectorSearch(v, e, !0);
    //待删除的元素不存在，删除失败
    if (pos < 0)
    {
        return -1;
    }
    return VectorRemoveByPos(v, pos);
}

//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
int VectorRemoveByPosU(VECTOR *v, int pos)
{
    if ((pos < 0) || (pos >= v->size) || VectorEmpty(v))
    {
        return -1;
    }
    void *target = (char *)v->elems + v->elemSize * pos;
    if (pos != (v->size - 1))
    {
        memmove(target, (char *)target + v->elemSize, v->elemSize * (v->size - pos - 1));
    }
    v->size --;
    if ((v->size * 2 <= v->capacity) && v->fSupportGrow)
    {
        VectorReduce(v);
    }
    return 0;
}

//删除元素（无需深度删除），返回值：!0--删除失败，0--删除成功
int VectorRemoveU(VECTOR *v, void *e)
{
    if (VectorEmpty(v))
    {
        return -1;
    }
    int pos = VectorSearch(v, e, !0);
    //待删除的元素不存在，删除失败
    if (pos < 0)
    {
        return -1;
    }
    return VectorRemoveByPosU(v, pos);
}

//更新元素
void VectorUpdate(VECTOR *v, int pos, const void *e)
{
    memcpy((char *)v->elems + pos * v->elemSize, e, v->elemSize);
}

//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 ;
}
```