Vector

查找- 顺序查找的实现， 平均复杂度为O(n)

```cpp
/*
Function: 通过值在数组中找到其位置.
@param elem: 需要查找的值
@return: 返回为对应的索引.
@throws std::invalid_argument: 如果找不到, 抛出此异常.
@example:
    ArrayList<int> al {1, 5, 12, 11};
        assert(al.location(5) == 1)
*/
template <typename ValueType>
size_t Vector<ValueType>::location(const ValueType &target) {

    for (int i = 0; i < size; i++) {
        if (array[i] == target)
            return i;
    }
    // 顺序查找的实现, 平均复杂度为 O(n)
    throw std::invalid_argument("Not find the element"
                                            "in the array");
}
```

查找- 二分

```cpp
/* 采用了二分查找, 因此效率为O(logn),使用此函数, 必须确保数组是有序的.*/
template <typename ValueType>

  size_t ArrayList<ValueType>::location(
                    ValueType target,
                    bool is_sorted) {

    assert(isSorted(array))
    int lo = 0;
    int hi = size - 1;

    while (lo <= hi) {
        int mid = lo + (hi - lo) / 2;

        if (array[mid] == target) {
            return mid;
        }
        else if (array[mid] > target) {
            hi = mid - 1;
        }
        else { // array[mid] < target
            lo = mid + 1;
```

```
        }
    }

    throw std::invalid_argument("the target is not"
                                " in this array");
}
```

LinkedList

头插法(操作head)

链表的头插法是指将新元素插入到链表的头部，成为新的头节点，而原来的头节点则成为新节点的后继节点。

头插法相比尾插法可以更快地实现链表的逆序 。

```
template<typename ValueType>
void LinkedList<ValueType>::add(ValueType val) {

     Cell *cp = new Cell(val, head);
     head = cp;
     count++;
}
```

尾插法(操作tail)

链表的尾插法是指将新元素插入到链表的尾部，成为新的尾节点，而原来的尾节点则成为新节点的前驱节点。尾插法是链表中最常见的插入方法之一。

```
template<typename ValueType>
void LinkedList<ValueType>::add(ValueType val) {

    if (head == nullptr) {
        head = new Cell(val, nullptr);
        tail = head;
        return;
    }

    tail->link = new Cell(val, nullptr);
    tail = tail->link;
    count++;
}
```

使用带头结点的指针的两个好处。

1. 简化逻辑判断

   如上述的尾插法代码，如果带头节点，意味着head永不为nullptr。

   ```
   template<typename ValueType>
   void LinkedList<ValueType>::add(ValueType val) {

       tail->link = new Cell(val, nullptr);
       tail = tail->link;
       count++;
   }
   ```

2．简化代码

带头节点的代码，能使 **真实的头节点** 存在前驱节点。

头节点的特殊性在于其没有前驱结点，所以代码中总需要判断是否是第一个结点。带头节点的可以简化代码。

```
    // 第一个节点是要删除的节点
    if (head→value == val) {
        Cell *temp = head;
        head = head→link;
        delete temp;
        count--;
        return;
    }

        // 第二个或者之后的节点是要删除的节点
    Cell *prev = head;
    Cell *curr = head→link;
```

QUEUE

关于head的取值问题?

在上述表示中，head拥有下一个将出队的队列中头元素的索引，而tail拥有队列末尾元素的索引。很显然，在一个空队列中tail域应该为0，它表示数组的初始位置，**但是head域的值为多少?**
为了方便，通常的策略也是设置head域的值也为0。当队列采用这种方法定义时，head和tail相等，并表示队列为空。

```
template<typename ValueType>
bool Queue<ValueType>::isEmpty() const
{
    return head_ == tail_;
}
```

取模运算

取模运算：逻辑上变成了无限长的队列， 尽管实际上会落在一个有限的范围里。

```
template<typename ValueType>
int Queue<ValueType>::size() const
{
    return (tail_ + capacity_ - head_) % capacity_;
}
```

QUEUE_LIST: 使用尾插法构造, FIFO

```
    void enquque(const_reference value) {
        // 由于在更新链表tail 所以说这是尾插法。
        Cell *cp = new Cell(value, nullptr);

        if (head_ == nullptr) {
            head_ = cp;
        } else {
            tail_->link = cp;
        }

        tail_ = cp;
        count_++;
    }
```

STACK

QUEUE_LIST: 使用头插法构造，FILO

```
template<typename value_type>
void Stack<value_type>::push(value_type value)
{
    Cell *cp = new Cell(value, head_);
    head_ = cp;
    count_++;
}
```

```
    void deepCopy(const Stack<ValueType> & src) {
        this->count_ = src.count_;
        if (src.isEmpty()) {
            clear(); // 这个实现不一定对?
            // 如果是拷贝赋值，会复数调用clear。
            return;
        }
        Cell *dst_list = nullptr;
        for (Cell *cp = src.head_->link;cp != nullptr; cp = cp->link) {
            Cell *ncp = new Cell(cp->data, nullptr);

            if (dst_list == nullptr) {
                head_ = ncp;
            } else {
                dst_list->link = ncp;
            }
            dst_list = ncp;
        }
    }
```