Chapter 5 VECTOR & DEQUE

VECTOR 向量

- vector，deque，list都为sequential-container（元素都有序）
- 向量判断条件
  - 条件1：支持随机访问，只要给了下标或者索引（index），然后可以访问对应位置只需要为**常量**的时间复杂度（in constant time）
  - 条件2：在序列的末端插入或删除一个元素只需要常量时间复杂度
- 使用时需声明头文件 #include<vector>
- 向量是SMART ARRAY
  1. 支持random-access 通过下标[ ]可返回对应位置
  2. size()能返回真正的元素个数 区分于向量所占存储空间
  3. 可自动扩充大小
  4. 可在任意位置插入删除
  5. 可赋值运算
  6. 属于模板类
  7. 和数组一样高效
- 向量的特点
  1. 具有构造性
  2. 赋值运算时会将一个向量的所有元素赋值给另一个向量

```
vector<t>& operator=(const vector<t>& x); //运算符重载

example.weights = oldweights;

//相当于实现以下效果 调用重载后的=
weights.operator=(oldweights);
```

- 向量的特点（续）
  3. 只要涉及元素的移动，最坏时间复杂度都为O(n)
  4. begin()为容器起始位置 end()为容器最后一个元素的下一个位置

- 向量的基础表现形式（物理存储结构）：数组

- 向量的逻辑结构
- start为开始元素的位置 finish为**最后一个元素的下一位置** end_of_storage为存储空间结束位置（即**最后一个存储单元的下一位置**）
  - begin()返回start
  - end()返回finish
  - size()返回finish-start
  - empty()返回判断start==finish
  - finish=end_of_storage时需要扩容
- 返回指针能自动构造迭代器
- 后端推入push_back即为finish所指位置

```
if（finish ！= end_of_storage) // 最坏时间复杂度为常量
{
    *finish = x;
    finish ++;
}
else // 需要扩容 复制到新的向量 最坏时间复杂度为O(n)
{
}

// 不等概率
n/(n+1)*O(1) + 1/(n+1)*O(n)
=
O(1)
// 因为扩容为小概率事件 从而总和变为常量事件（时间平均的方式 - amortizedtime）
// 因此摊销时间复杂度为常量
```

- insert插入操作需要考虑向量的存储空间是否已满（是否需要扩容）
- 若不需要扩容，需要考虑插入位置是否为最后一个元素位置，若是则相当于执行push_back操作，若不是则需要执行insert的后向复制操作
- itr需要回到原来的位置，因为扩容了新向量，新的itr指向扩容之后复制的新位置，导致itr的实际位置可能会变，所以将itr赋值到新的指向元素

- position == end() 相当于push_back

High-precision Arithmetic 高精度计算

key facts:

1. **to generate a very long integer that is prime, averagetime(n) is o((log n)^3).** 生成素数整数
   if $n = 10^{200}$, $(\log_{10} n)^3 = 200^3 = 8{,}000{,}000$.

2. **to factor a very long integer that is not prime, averagetime(n) is o($10^{n/2}$).** 因式分解 非素数整数

- 实现长整数类 very_long_int

```
verylong1 + verylong2
//实际调用以下代码
verylong.operator+(verylong2)
```

- digit_char - '0' 相当于减去'0'对应的ASCII码48 从而得到数值
- 注意数组的**高位（array[0]为高位）**和实际长整数运算的**低位（array[size-1]为最后一位）**问题 最后一位（size-1） 倒数第i位即为（size-1-i）
- 记得最后要reverse 因为push_back后所得的实际为倒过来的
- 还要注意进位的1是否保留

DEQUE 双端队列

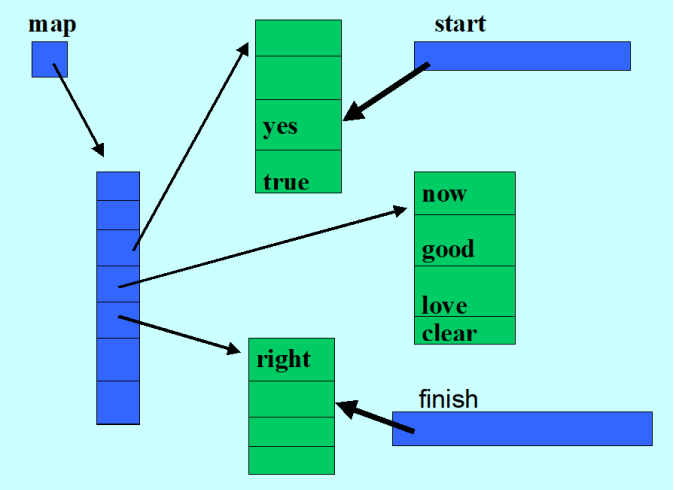
- 与向量类非常相似
  - 具有随机访问性

    - 共用操作方法基本相同 除了capacity和reserve
    - 相同接口
- 可理解为两端都开放的向量
    - 在**首端和末端**的插入和删除操作都只需要花费**常量时间**
    - 增加了push_front和pop_front方法
- 删除前端元素：向量vector只能用erase（迭代器）；双端队列deque可使用push_front
    - worstTime is O(n);
    - averageTime is constant;
    - amortizedTime(n) is constant）
- 运行时vector更快，但当涉及大量靠近前端插入和删除的操作时，deque更具优势

映射数组

deque类中含有map字段；start和finish字段为迭代器

- map是指向array的指针
- array包含指向block的指针
- 块block里放着项item
- start指向队列第一个项的位置
- finish指向队列最后一项之后的位置



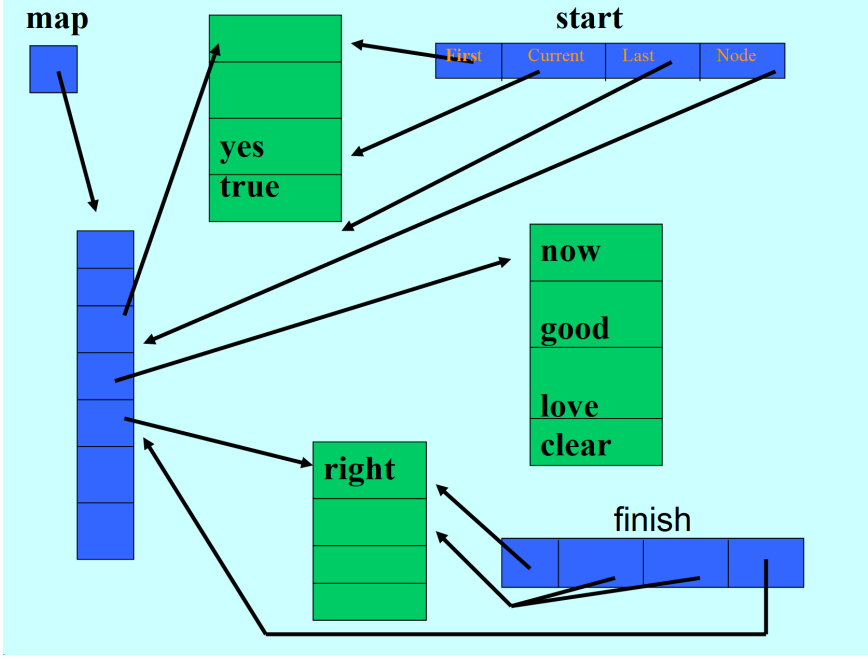
- 映射数组上的每个元素对应一个块 块中包含元素的数量固定

▼ 嵌套在deque类的iterator类有四个域

前提：一个迭代器位于项（x的位置

- first指针：指向包含项x的block的第一个项
- current指针：指向项x的位置
- last指针：指向包含项x的block的尾部的下一个单元
- node指针：指向map中单元的指针，其中map指向包含项x的block起始地址（即表达这一块的映射处的元素地址- 该元素也是指针 即指针的指针）
  - *node = start.first：node的吸取和start中first的值为相同地址

⭐注意此处虽然*node和start指向地址相同，但两者的实际意义和操作对象不同，node在映射数组中移动，start在块中移动。



- start和finish是固有的迭代器
- start指向整个双端队列最开始的元素 所以current永远指向block中的**第一个元素**
- finish永远指向**最后一个元素之后的位置**

  e.g finish中的current指向right元素的**下一位置**

```
//push_front(x);
--start.current;
* (start.current) = x;

//push_back(x)
(*finish.current) = x;
++ finish. current;

//扩容
*(--node) = new block; //找到前一块的地址
start.first = *(node);
start.current = *(node) + block_size - 1;
start.last = *(node) + blocksize;
*(start.current) = m; //插入新值
```

双端队列扩容 resize

- 当映射数组满了时需要扩容
- 只需对映射数组扩容 相应的映射块无需复制 新扩容的数组能映射回原来块的地址
- 扩容的最坏时间复杂度仍为**O(n)**：由映射数组的元素造成（n/4）

迭代器自增

用**last指针**的位置来判断current**是否离开该block位置**

再利用node找到映射数组对应元素的下一元素位置

```
if(itr.current == itr.last){ //若迭代器当前指向位置为last位置 相当于迭代器已知道最后一个元素的下一位置（因为last指向的就是block下一单元的位置） 故此时迭代器指针已指在block块之外
    itr.first = *(++itr.node);
    itr.current = itr.first;
    itr.last = itr.first + block_size;
}
```

随机访问 random access

通过下标index访问对应block（在第几块）和offset（块中的第几个位置）

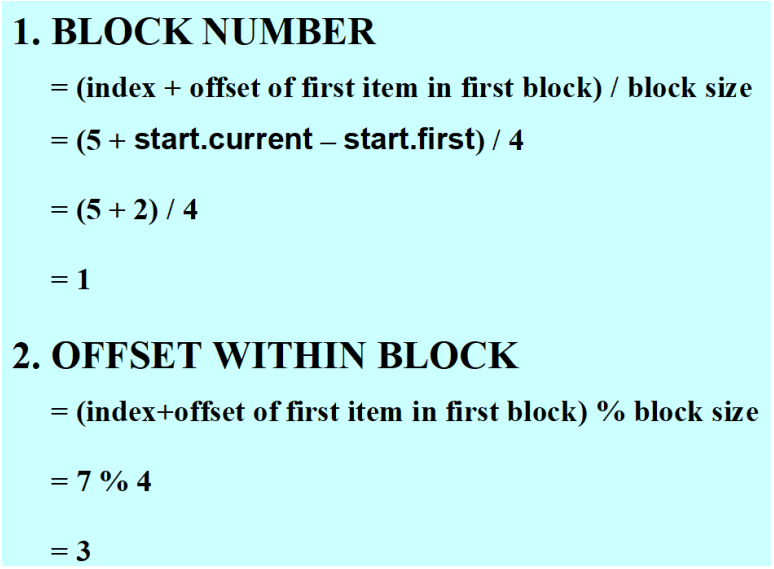
空的元素也包含在内 用**(current-first)**得到空的元素的个数

```
(4 + (start.current - start.first)) //4为该block总共有4个元素
```

step 1：首先将所求位置下表index加上每一个block的元素个数x

step 2：通过**x整除/=n**得到所在块数n

step 3：通过**x取余%=m**得到在第n+1块的第m个元素



补充

- 对于deque内部某个索引i处的插入或删除操作，移动的项数是minimum(i, length-i)

For example, to insert at words [5], the number of items moved is 7 – 5 = 2.

To delete the item at index 1, the number of items moved is 1.