unit5

practice1

1. 线性表是否为空的判断，返回 true 或 false
2. 线性表中元素的数量，返回一个非负数
3. 具体的元素，返回第 0 个元素、第 2 个元素、第 6 个元素、按照具体的实现（可能是一个异常）
4. 元素的索引，返回元素 a、c、q 的索引
5. 删除索引值对应的元素，不返回任何值
6. 在要求的索引上插入对应的元素，没有返回值

practice2

初始状态：
a b c d e _ _ _ _ _

insert(0, f)
f a b c d e _ _ _ _

insert(3, g)
f a b g c d e _ _ _

insert(7, h)
f a b g c d e h _ _

erase(0)
a b g c d e h _ _ _

erase(4)
a b g c e h _ _ _ _

practice3

见 code/includes/changeLength2D.h

practice4

略

practice5-18

见 code/includes/arrayList.h

---

practice19

1. 从均摊复杂度角度来看，参考 Why is vector array doubled?
   设操作 inserts 与 others 数量分别为 n, x. 则有 $O_{diff} = O(c^{\lfloor log_c n \rfloor}) \Rightarrow O_{avg} = \frac{O_{diff}}{x+n} \approx O(1)[x = 0, \lfloor log_c n \rfloor = log_c n]$, 可见从均摊角度看, 成倍增加数组长度不会影响每次 insert 的时间复杂度

2. 不管是看中文版还是英文版[1]，我个人感觉(poor english)还是应该从所有操作的总耗时上比较(就像文中使用的 `increase by a constant factor` 体现在数组长度**成倍**增加上，那么 `increases by at most a constant factor`也应该体现在比值上)
   设数组容量足够大时的所有操作(n+x)总耗时为 O_{total}，扩容耗时为 O_{diff}，则有 $\frac{O_{total}+O_{diff}}{O_{total}} = \frac{O_{total}+O(n)}{O_{total}} \leq 2$
   当 $x = 0, \lfloor log_c n \rfloor = log_c n$时， $\frac{O_{total}+O_{diff}}{O_{total}} = 2$

注:

1. 扩容的时间复杂度为 $\Theta(n)$，证明过程见书本，此处略
2. 均摊角度中 O_{diff} 不需考虑移动数组成本，总耗时角度 O_{total} 包含移动数组成本

---

practice20

1. 见 code/includes/arrayList.h
2. 证:
   相比于不需要改变容量，需要改变容量时增加的复杂度体现在改变数组所需时间上(即复制元素到新数组的时间)，为简便描述，以下描述的都是这部分耗时。
   当线性表数量降至 arrayLength/4 以下才触发缩容，因此 $Max_{缩容次数}$ = 扩容次数 - 1

   首先需要找到最耗时情况。假设扩容到 2^k (2为常数因子)时，后有 x 次扩容，y 次缩容。若先扩容完再缩容，则共有复制次数

$$\underbrace{2^k + 2^{k+1} + \cdots + 2^{k+x-1}}_{x次扩容} + \underbrace{2^{k+x-2} + 2^{k+x-3} + \cdots + 2^{k+x-1-y}}_{y次缩容} \tag{1}$$

   若中间有先缩容再扩容的情况，那么扩容的复制次数和缩容的复制次数都少了，因此最耗时情况为**先扩容完再缩容**
   由 (1) 可以得到

$$O_{diff} = O_{扩容}(m) + O_{缩容}(p) < 2O_{扩容}(m) \quad p < m < n$$

   令 n 次操作均为插入操作，根据定理 5.1 可以进一步得到更宽泛的边界

$$O_{diff} < 2O_{扩容}(m) < 2O(n) = g(n) \cdot O(1) \quad m < n$$

因此，总操作数为

$$f(n) + g(n) = cf(n)$$

practice21

参考 20 2)，将扩容因子 2 替换为 c

---

practice22

1. 见 code/includes/arrayList.h
2. 见 code/test/arrayList 中的 testReverse()
3. 见 code/test/arrayList 中的 testReverse()
4. 见 code/includes/arrayList.h
5. 见 code/includes/arrayList.h
6. 修改 2 即可，略

---

practice23

见 code/includes/arrayList.h

---

practice24

1. origin: {123456789}, leftShift(3), result: {456789123}

operation	status
reverse(0,8)	987654321
reverse(0,5)	456789321
reverse(6,8)	456789123

2. 见 code/includes/arrayList.h

---

practice25

1. 见 code/includes/arrayList.h
2. 证明：只遍历一遍，其中操作都为 O(1) (假设调用析构函数和复制时间相同)

---

practice26

见 code/includes/arrayList.h

---

practice27

见 code/includes/arrayList.h

---

practice28

见 code/includes/arrayList.h

---

practice29

见 code/includes/arrayList.h

practice30

见 code/includes/arrayList.h

---

practice 31

见 code/includes/circularArrayList.h

---

practice 32

见 code/includes/circularArrayList.h

---

practice 33

见 code/includes/circularArrayList.h

---

practice 34

见 code/includes/circularArrayList.h

---

practice 35

见 code/includes/circularArrayList.h

---

practice 36

见 code/includes/circularArrayList.h

---

practice 37

见 code/includes/vectorList.h

practice 38

见 code/includes/vectorList.h

practice 39

见 code/includes/vectorList.h

practice 40

见 code/includes/vectorList.h

practice 41

见 code/includes/multiList.h

practice 42

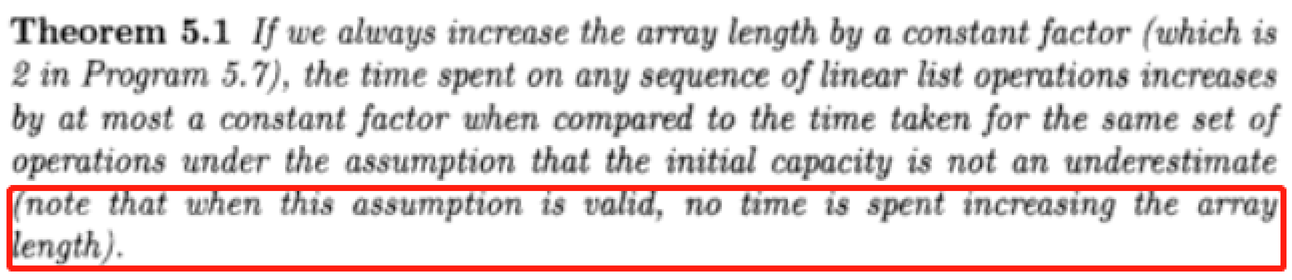
见 code/includes/multiList.h

practice 43

见 code/includes/multiList.h

practice 44

略



Theorem 5.1 *If we always increase the array length by a constant factor (which is 2 in Program 5.7), the time spent on any sequence of linear list operations increases by at most a constant factor when compared to the time taken for the same set of operations under the assumption that the initial capacity is not an underestimate (note that when this assumption is valid, no time is spent increasing the array length).*

1. ↩