## Динамические массивы

У списков есть свои неоспоримые преимущества, но есть и один недостаток -- мы не можем обратиться к конкретному элементу быстро, по индексу, одним обращением. Сложность операции выборки i-го элемента, очевидно, O(n) -- нам надо, в наихудшем случае, пробежаться по всему списку с начала до конца. А массивы через индексирование обычно предоставляют меру O(1) -- то есть не зависящую от размера массива. И мы при этом хотим работать с массивами, как с динамическими объектами -- чтобы они автоматически расширялись при добавлении в них элемента.

В Python имеется тип данных "список" (list), который в принципе полностью отвечает требованиям динамического массива. Список индексируется, позволяет добавлять элементы в позицию по индексу, автоматически расширяется когда в него добавляются новые элементы, и сжимается при удалении элементов.

**1. Схемы реаллокации**

Нас интересует низкоуровневая реализация механизма динамических массивов. Ключевой момент здесь -- эффективность. Менять реальный размер памяти, отводимый под массив, каждый раз, когда количество его элементов меняется, весьма накладно, так как каждый раз придётся делать системный вызов. Поэтому обычно применяется компромиссный вариант: внутри динамического массива существует буфер, например на N элементов, и только когда добавляется новый N+1 элемент, размер этого буфера увеличивается с приличным запасом.

Схем увеличения буфера (реаллокации массива) две: аддитивная и мультипликативная.

**Аддитивная схема** обычно применяется, когда размер массива небольшой, и расходы на реаллокацию невелики. В таком случае экономичнее увеличивать его размер относительно небольшими фиксированными порциями. **Мультипликативная схема** подразумевает достаточно сильное изменение размера -- в разы, так как реаллокация крупного массива потребует существенных ресурсов. Обычно он удваивается. Пример мультипликативной схемы, принятой в стандартной библиотеке Java:

size = (size \* 3) / 2 + 1

Часто применяется и отношение "золотого сечения" 1,618.

В целом, политика регулирования такого размера сильно зависит от конкретной решаемой задачи.

**Важно**. В тестах используется схема, когда увеличение буфера происходит в два раза, а уменьшение в полтора раза (текущее значение размера буфера делится на 1.5, и результат приводится к целому типу, никаких округлений!). При этом сохраняем минимальную ёмкость 16 элементов.   
Увеличение буфера выполняем, когда он весь полностью заполнен, и выполняется попытка добавления. Сокращение буфера выполняем, когда его заполненность после операции удаления станет строго меньше, чем заданный процент заполнения. В тестах используйте этот процент равным 50%.   
Придерживайтесь этой схемы в своём коде для успешного тестирования.

**2. Анализ мультипликативной схемы с помощью банковского метода**

В общем случае недостаток мультипликативной схемы в том, что размер массива будет увеличиваться в геометрической прогрессии, а желаемая сложность O(1) в реальности будет o(1) (в среднем). Если вставка попадёт на реаллокацию, задержка времени может быть существенной. Поэтому в противовес так называемому **агрегированному методу** анализа сложности (и, соответственно выбору метода реаллокации), который подразумевает вычисление итоговой суммарной оценки всех операций (общее время работы) применяют **банковский метод**.   
Его идея в том, что дешёвые операции мы делаем чуть дороже, чем они реально стоят, а дорогие наоборот подешевле. При этом удаётся поддерживать всё время положительный баланс, несмотря на то, что дорогие операции по абсолютной цене могут уводить его в минус.   
Например, каждый случай вставки элемента, когда реаллокации не происходит, и каждый случай копирования элемента при копировании, можно оценивать не в одну условную операцию записи, а в три. Как только накапливается достаточная сумма, чтобы оплатить очередную реаллокацию, она выполняется. Причём цена её зависит от количества элементов в массиве и выбирается например, как степень двойки, не превышающая новый размер буфера.

**3. Реализация динамического массива в Python**

Мы смоделируем работу динамического массива в Python с помощью изученной в прошлом занятии библиотеки ctypes, которая позволяет работать напрямую с индексируемым массивом объектов PyObject.

В классе DynArray потребуются три поля: count -- текущее количество элементов в массиве, capacity -- текущая ёмкость буфера (исходно 16 единиц), и array -- по сути, указатель на блок памяти нужной ёмкости, хранящий элементы PyObject.

**class** **DynArray**:

**def** **\_\_init\_\_**(self):

self.count = **0**

self.capacity = **16**

self.array = self.make\_array(self.capacity)

**def** **\_\_len\_\_**(self):

**return** self.count

Формирование такого блока выполняет метод make\_array().

**def** **make\_array**(self, new\_capacity):

**return** (new\_capacity \* ctypes.py\_object)()

Стандартный метод \_\_getitem\_\_() обеспечивает поддержку индексации нашего класса. В неё мы встроим проверку корректности индекса в рамках границ, и генерацию соответствующего исключения.

**def** **\_\_getitem\_\_**(self,i):

**if** i < **0** **or** i >= self.count:

**raise** **IndexError**('Index is out of bounds')

**return** self.array[i]

Ключевой метод resize(), меняющий размер внутреннего буфера, просто формирует через make\_array() новый буфер, и копирует в него текущее содержимое.

**def** **resize**(self, new\_capacity):

new\_array = self.make\_array(new\_capacity)

**for** i **in** range(self.count):

new\_array[i] = self.array[i]

self.array = new\_array

self.capacity = new\_capacity

Наконец, метод append() добавляет новый элемент в конец массива.

**def** **append**(self, itm):

**if** self.count == self.capacity:

self.resize(**2**\*self.capacity)

self.array[self.count] = itm

self.count += **1**

Проверяем:

da = DynArray()

**for** i **in** range(**64**):

da.append(i)

print (da[i])

[[ предыдущее занятие ]](http://skillsmart.ru/algo/py-kf32y/y6a342a5104u.html)

**4. Задания.**

4.1. Добавьте метод insert(i, itm), который вставляет в i-ю позицию объект itm, сдвигая вперёд все последующие элементы. Учтите, что новая длина массива может превысить размер буфера.

4.2. Добавьте метод delete(i), который удаляет объект из i-й позиции, при необходимости сжимая буфер.

В обоих случаях, если индекс i лежит вне допустимых границ, генерируйте исключение.

Важно, единственное исключение: для метода insert() параметр i может принимать значение, равное длине рабочего массива count, в таком случае добавление происходит в его хвост.

4.3. Оцените меры сложности для этих двух методов.

4.4. Напишите тесты, проверяющие работу методов insert() и delete():  
-- вставка элемента, когда в итоге размер буфера не превышен (проверьте также размер буфера);  
-- вставка элемента, когда в результате превышен размер буфера (проверьте также корректное изменение размера буфера);  
-- попытка вставки элемента в недопустимую позицию;  
-- удаление элемента, когда в результате размер буфера остаётся прежним (проверьте также размер буфера);  
-- удаление элемента, когда в результате понижается размер буфера (проверьте также корректное изменение размера буфера);  
-- попытка удаления элемента в недопустимой позиции.

## заготовка класса для автоматической проверки

**import** **ctypes**

**class** **DynArray**:

**def** **\_\_init\_\_**(self):

self.count = **0**

self.capacity = **16**

self.array = self.make\_array(self.capacity)

**def** **\_\_len\_\_**(self):

**return** self.count

**def** **make\_array**(self, new\_capacity):

**return** (new\_capacity \* ctypes.py\_object)()

**def** **\_\_getitem\_\_**(self,i):

**if** i < **0** **or** i >= self.count:

**raise** **IndexError**('Index is out of bounds')

**return** self.array[i]

**def** **resize**(self, new\_capacity):

new\_array = self.make\_array(new\_capacity)

**for** i **in** range(self.count):

new\_array[i] = self.array[i]

self.array = new\_array

self.capacity = new\_capacity

**def** **append**(self, itm):

**if** self.count == self.capacity:

self.resize(**2**\*self.capacity)

self.array[self.count] = itm

self.count += **1**

**def** **insert**(self, i, itm):

# добавляем объект itm в позицию i, начиная с 0

**def** **delete**(self, i):

# удаляем объект в позиции i