Source : http://www.laurentluce.com/posts/python-list-implementation/

Lists in Python are powerful and it is interesting to see how they are implemented internally.

Following is a simple Python script appending some integers to a list and printing them.

```
>>> l = []
>>> l.append(1)
>>> l.append(2)
>>> l.append(3)
>>> l
[1, 2, 3]
>>> for e in l:
...   print e
...
1
2
3
```

As you can see, lists are iterable.

List object C structure

A list object in CPython is represented by the following C structure. ob_item is a list of pointers to the list elements. allocated is the number of slots allocated in memory.

```
typedef struct {
    PyObject_VAR_HEAD
    PyObject **ob_item;
    Py_ssize_t allocated;
} PyListObject;
```

List initialization

Let's look at what happens when we initialize an empty list. e.g. l = [].

```
arguments: size of the list = 0
returns: list object = []
PyListNew:
    nbytes = size * size of global Python object = 0
    allocate new list object
    allocate list of pointers (ob_item) of size nbytes = 0
    clear ob_item
    set list's allocated var to 0 = 0 slots
    return list object
```

It is important to notice the difference between allocated slots and the size of the list. The size of a list is the same as len(l). The number of allocated slots is what has been allocated in memory. Often, you will see that allocated can be greater than size. This is to avoid needing calling realloc each time a new elements is appended to the list. We will see more about that later.

Append

We append an integer to the list: l.append(1). What happens? The internal C function app1() is called:

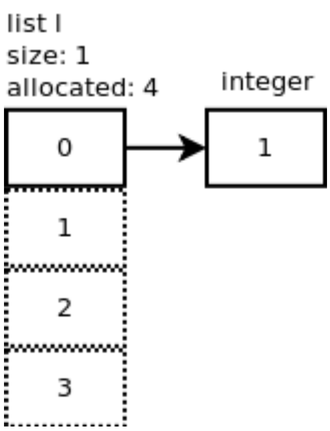
```
arguments: list object, new element
returns: 0 if OK, -1 if not
app1:
    n = size of list
    call list_resize() to resize the list to size n+1 = 0 + 1 = 1
    list[n] = list[0] = new element
    return 0
```

Let’s look at list_resize(). It over-allocates memory to avoid calling list_resize too many time. The growth pattern of the list is: 0, 4, 8, 16, 25, 35, 46, 58, 72, 88, …

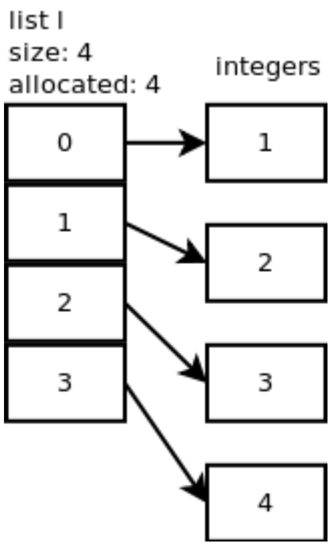
```
arguments: list object, new size
returns: 0 if OK, -1 if not
list_resize:
    new_allocated = (newsize >> 3) + (newsize < 9 ? 3 : 6) = 3
    new_allocated += newsize = 3 + 1 = 4
    resize ob_item (list of pointers) to size new_allocated
    return 0
```

4 slots are now allocated to contain elements and the first one is the integer 1. You can see on the following diagram that l[0] points to the integer object that we just appended. The dashed squares represent the slots allocated but not used yet.

Append operation amortized complexity is O(1).



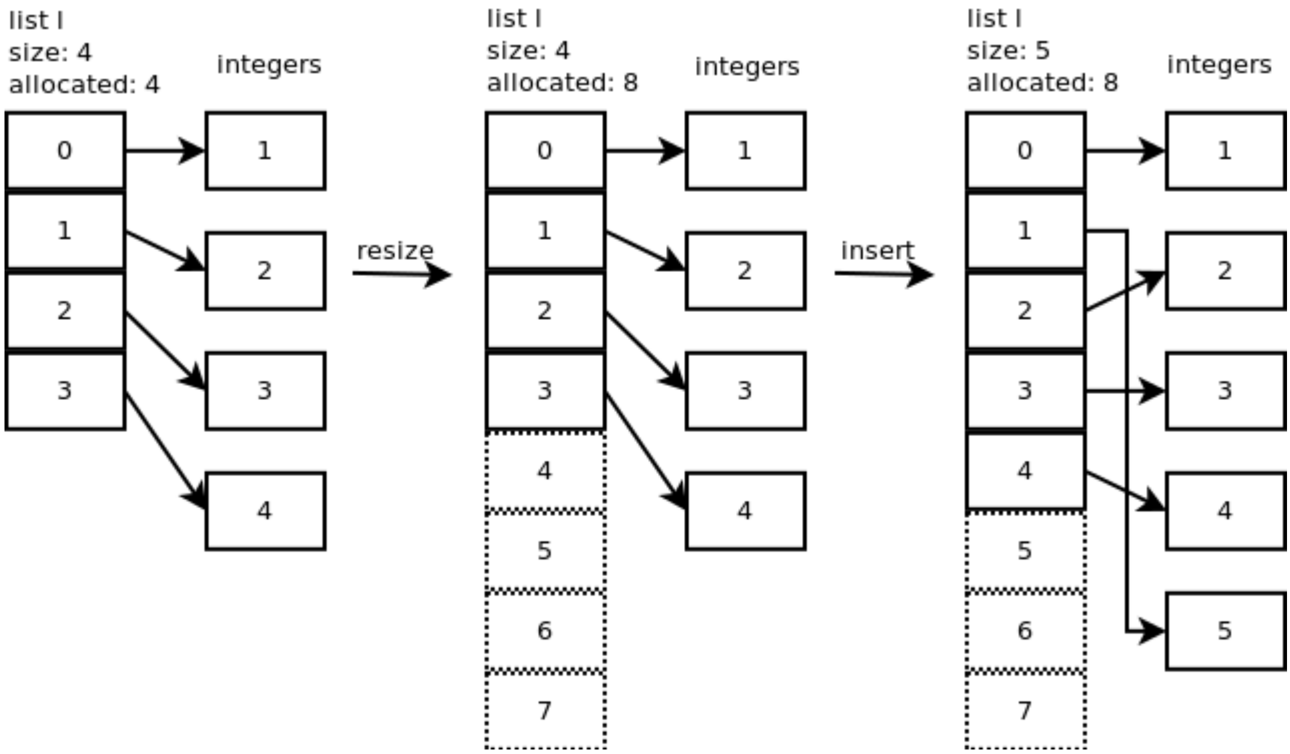
We continue by adding one more element: l.append(2). list_resize is called with n+1 = 2 but because the allocated size is 4, there is no need to allocate more memory. Same thing happens when we add 2 more integers: l.append(3), l.append(4). The following diagram shows what we have so far.



Insert

Let’s insert a new integer (5) at position 1: l.insert(1,5) and look at what happens internally. ins1() is called:

```
arguments: list object, where, new element
returns: 0 if OK, -1 if not
ins1:
    resize list to size n+1 = 5 -> 4 more slots will be allocated
    starting at the last element up to the offset where, right shift each element
    set new element at offset where
    return 0
```



The dashed squares represent the slots allocated but not used yet. Here, 8 slots are allocated but the size or length of the list is only 5.

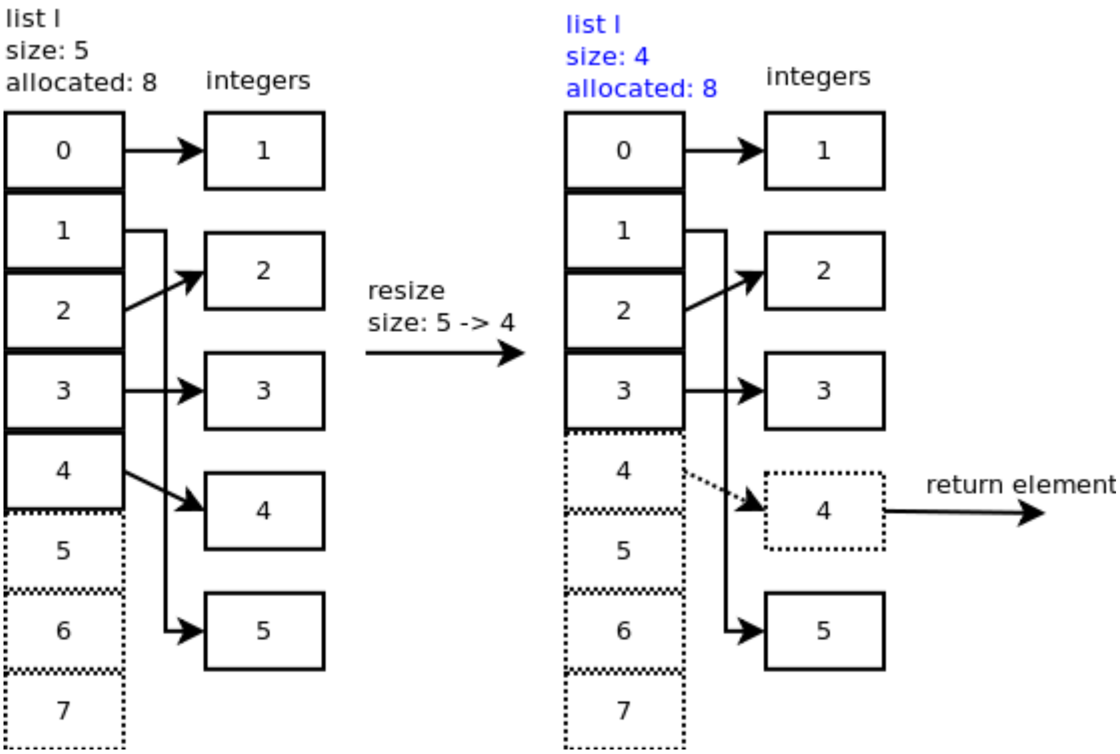
Insert operation complexity is O(n).

Pop

When you pop the last element: l.pop(), listpop() is called. list_resize is called inside listpop() and if the new size is less than half of the allocated size then the list is shrunk.

```
arguments: list object
returns: element popped
listpop:
    if list empty:
        return null
    resize list with size 5 - 1 = 4. 4 is not less than 8/2
so no shrinkage
    set list object size to 4
    return last element
```

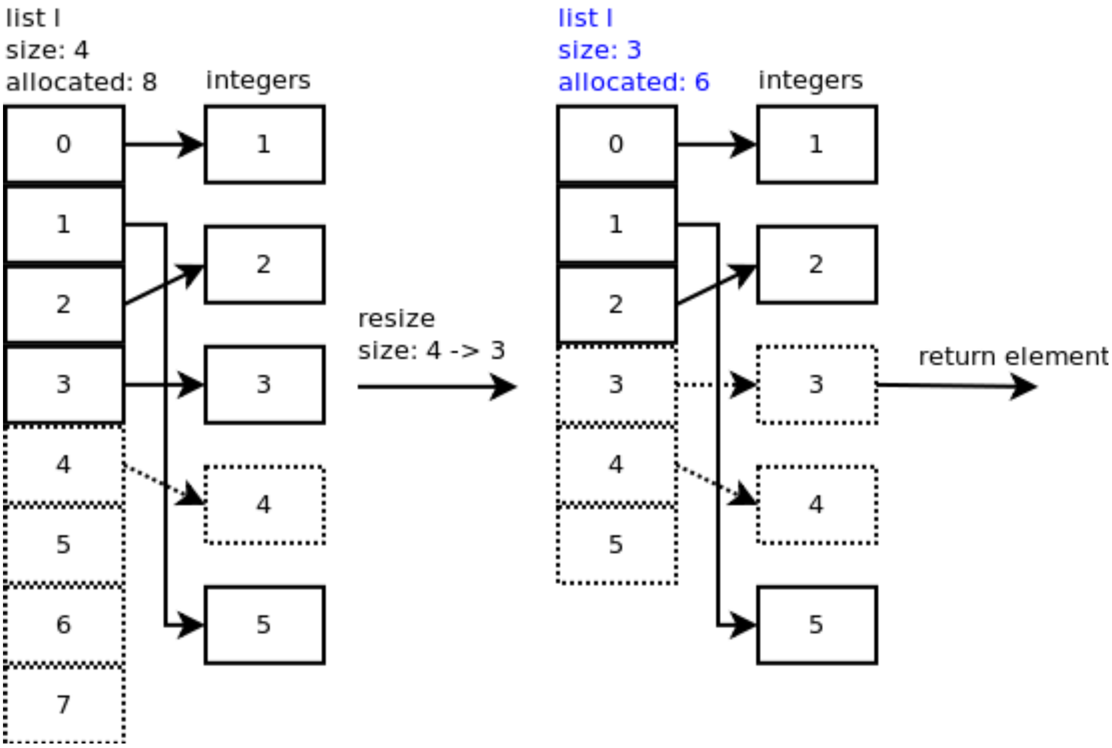
Pop operation complexity is O(1).



You can observe that slot 4 still points to the integer but the important thing is the size of the list which is now 4.

Let's pop one more element. In list_resize(), size – 1 = 4 – 1 = 3 is less than half of the allocated slots so the list is shrunk to 6 slots and the new size of the list is now 3.

You can observe that slot 3 and 4 still point to some integers but the important thing is the size of the list which is now 3.



Remove

Python list object has a method to remove a specific element: l.remove(5). listremove() is called.

```
arguments: list object, element to remove
returns none if OK, null if not
listremove:
    loop through each list element:
        if correct element:
            slice list between element's slot and element's slot + 1
            return none
    return null
```

To slice the list and remove the element, list_ass_slice() is called and it is interesting to see how it works. Here, low offset is 1 and high offset is 2 as we are removing the element 5 at position 1.

```
arguments: list object, low offset, high offset
returns: 0 if OK
list_ass_slice:
    copy integer 5 to recycle list to dereference it
    shift elements from slot 2 to slot 1
    resize list to 5 slots
    return 0
```

Remove operation complexity is O(n).