# Definition

List is a very important data structure in Python programming language. I have been using list to implement most of the data structures in the collection file. Python list has different types of implementation namely CPython and JPython. While in CPython list is constructed by using a list of pointers to the list of elements (this is a dynamic array in which size can be modified dynamically), JPython uses an array list.

A list is represented by using the object C structure

|  |  |
| --- | --- |
| 1 | typedef struct { |

|  |  |
| --- | --- |
| 2 | PyObject\_VAR\_HEAD |

|  |  |
| --- | --- |
| 3 | PyObject \*\*ob\_item; |

|  |  |
| --- | --- |
| 4 | Py\_ssize\_t allocated; |

|  |  |
| --- | --- |
| 5 | } PyListObject; |

Allocated is the number of slots allocated in the memory, and it size can be grown when needed.

One thing we need to point out here is the allocated slots and the actual size of the list. These two numbers are different. Allocated slots are the number of slots registered in the memory, while the size is the actual number of elements are in the list. Based on this, the allocated is often greater than the actual size.

# Methods

## Append:

The append method must have the linear time operation. In C code, the app1() function is called. This function first will call another function list\_resize() to resize the array then assign the n element to be the value we want to append.

|  |  |
| --- | --- |
| 1 | arguments: list object, new element |

|  |  |
| --- | --- |
| 2 | returns: 0 if OK, -1 if not |

|  |  |
| --- | --- |
| 3 | app1: |

|  |  |
| --- | --- |
| 4 | n = size of list |

|  |  |
| --- | --- |
| 5 | call list\_resize() to resize the list to size n+1 = 0 + 1 = 1 |

|  |  |
| --- | --- |
| 6 | list[n] = list[0] = new element |

|  |  |
| --- | --- |
| 7 | return 0 |
|  |  |

The list\_resize() function will control how the list will grow, and the pattern is 0, 4, 8, 16, 25, 35, 46, 58 ,72 , 88...

|  |  |
| --- | --- |
| 1 | arguments: list object, new size |

|  |  |
| --- | --- |
| 2 | returns: 0 if OK, -1 if not |

|  |  |
| --- | --- |
| 3 | list\_resize: |

|  |  |
| --- | --- |
| 4 | new\_allocated = (newsize >> 3) + (newsize < 9 ? 3 : 6) = 3 |

|  |  |
| --- | --- |
| 5 | new\_allocated += newsize = 3 + 1 = 4 |

|  |  |
| --- | --- |
| 6 | resize ob\_item (list of pointers) to size new\_allocated |

|  |  |
| --- | --- |
| 7 | return 0 |
|  |  |

## Insert

Insert routine will help us insert a value to anywhere in the list. It first will assign the index pointer to our element then increment all other pointers.

|  |  |
| --- | --- |
| 1 | arguments: list object, where, new element |

|  |  |
| --- | --- |
| 2 | returns: 0 if OK, -1 if not |

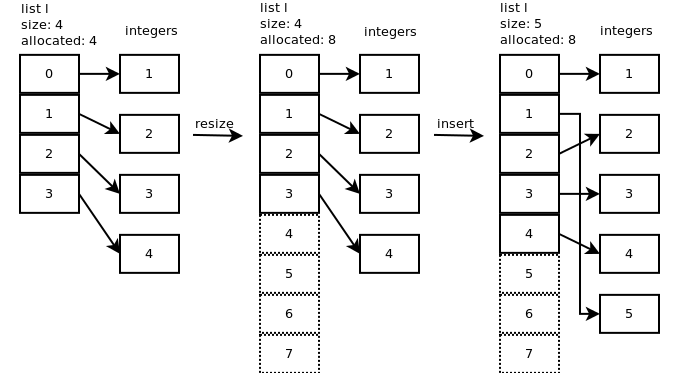
|  |  |
| --- | --- |
| 3 | ins1: |

|  |  |
| --- | --- |
| 4 | resize list to size n+1 = 5 -> 4 more slots will be allocated |

|  |  |
| --- | --- |
| 5 | starting at the last element up to the offset where, right shift each element |

|  |  |
| --- | --- |
| 6 | set new element at offset where |

|  |  |  |
| --- | --- | --- |
| 7 | return 0 |  |

.

The time complexity of this method is O(n)

## Pop

The pop function will first call the list\_resize(0 if the new size is less than half of the allocated size then the list is shrunk.

|  |  |
| --- | --- |
| 1 | arguments: list object |

|  |  |
| --- | --- |
| 2 | returns: element popped |

|  |  |
| --- | --- |
| 3 | listpop: |

|  |  |
| --- | --- |
| 4 | if list empty: |

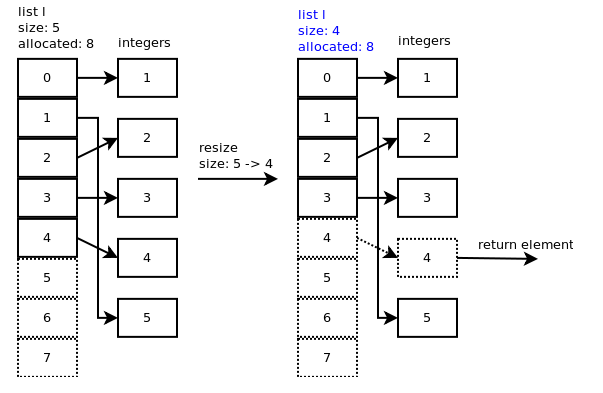
|  |  |
| --- | --- |
| 5 | return null |

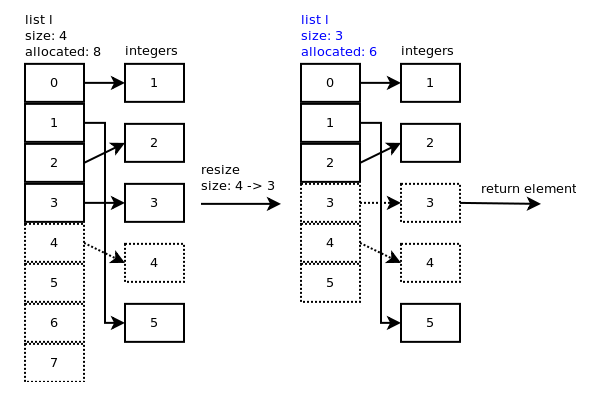
|  |  |
| --- | --- |
| 6 | resize list with size 5 - 1 = 4. 4 is not less than 8/2 so no shrinkage |

|  |  |
| --- | --- |
| 7 | set list object size to 4 |

|  |  |
| --- | --- |
| 8 | return last element |

Pop() with any argument will take 0(1) time complexity but the new python version support pop by index, this will take 0(n) time complexity





We can observe that after removing the allocated slot still have the pointer to the value, but now size of the list is only three, and we do not care about those value anymore.

## Remove

|  |  |
| --- | --- |
| 1 | arguments: list object, element to remove |

|  |  |
| --- | --- |
| 2 | returns none if OK, null if not |

|  |  |
| --- | --- |
| 3 | listremove: |

|  |  |
| --- | --- |
| 4 | loop through each list element: |

|  |  |
| --- | --- |
| 5 | if correct element: |

|  |  |
| --- | --- |
| 6 | slice list between element's slot and element's slot + 1 |

|  |  |
| --- | --- |
| 7 | return none |

|  |  |
| --- | --- |
| 8 | return null |
|  |  |

To slice the list and remove the list\_ass\_slice is called. Time complexity of this method is O(n)

|  |  |
| --- | --- |
| 1 | arguments: list object, low offset, high offset |

|  |  |
| --- | --- |
| 2 | returns: 0 if OK |

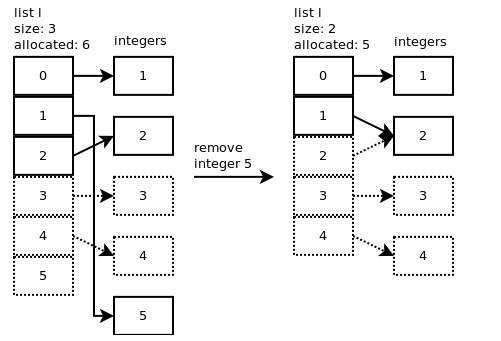
|  |  |
| --- | --- |
| 3 | list\_ass\_slice: |

|  |  |
| --- | --- |
| 4 | copy integer 5 to recycle list to dereference it |

|  |  |
| --- | --- |
| 5 | shift elements from slot 2 to slot 1 |

|  |  |
| --- | --- |
| 6 | resize list to 5 slots |

|  |  |
| --- | --- |
| 7 | return 0 |



Finally is the time complexity benchmark by Python document

