List

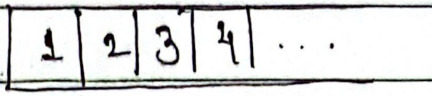
Array List Linked List

Date : .....

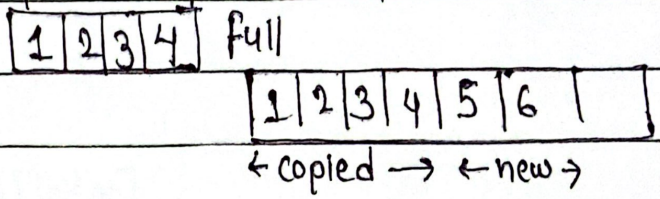
Page : .....

Array List: Implementation

Implemented in a memory with the help of Dynamic array.



If dynamic array gets fully occupied, then a larger size dynamic array is allocated. Then all elements are copied. A farther new element is added



If we have to insert element in between:
we have to push all the elements in the worst case.

When ever we have to insert element in beth the elemen time complexity becomes O(n) because we have to push n no. of element to make room for new one.

Time Complexity: O(n) ← Insertion

Space O(1) ← Search : on one operations

Array is a contiguous block of memory (bit addressable) all packed one next to another - no gaps

Benifits: Using an array based list is we can access any element of the list in O(1) time.

Date: .....
Page: .....

names = [ ] when you even initialize empty list
get · sizeof(names) → it occupies space.

In memory, what happens?
it creates Object header (24 bytes)
↳ sth that python uses internally
Other part - length of list stored as integer.
Python keeps track of length of list

Address	Value
0x00000	<OBJECT HEADER>
0x00008	
0x000010	length : int (8)
0x000018	
20	
30	

- When you call len(names) the length is returned.
- When a list is instantiated in memory. it pre-allocate a laeger block of memory for the list in anticipation go will add data to it.

What happens when we append an items to the list

- First thing, string "Alice" is created somewhere in memo.
names · append("Alice")
we don't know where in memory, Python goes to its ship and says " Hey user want string called 'Alice'.

- When you append 'Alice', you don't actually appent 'Alice' actual value, but the reference to Alice that lives in memory - 0x8d6fgaD

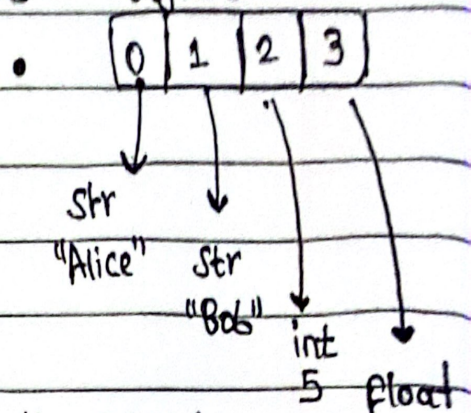
pythontutor.com

Date: .....

Page: .....

```
names = []
names.append('Alice')
names.append('Bob')
names.append(5)
names.append(3.15)
```

Frames Objects



Python list's data structure dictates the O(f(n)) of its operations.

Big-O	Code	Operations
O(1)	list[index]	get item by index
O(1)	list[index] = x	set object at index to be x
O(1)	list.append(x)	append x to the end of list
O(1)	list.pop()	remove from end of list & return it
O(n)	list.pop(i)	remove item at index i & return it.
O(n)	list.remove(x)	remove the first item whose value is x
O(n)	list.insert(i,x)	insert x at index i
O(n)	iteration (for x in list)	iterate over each item in list

Date :
Page :

* x = list[2] O(1) - as you can directly pick element of index mentioned.

list[0] ⇒ is at address 32 (24(header) + 8(length) + 0)

* Writing by Index:

list[2] = 'Eugen' → How to know where it goes in a memory?

24 + 8 + 8 * i ← index

head ↑ len ↑ reference
int

* Append:

list.append('Bart') → what happens? it gets added at the last of the items.

[memory add. of my list + 24 (obj header) + 8 (length) + index (8 * length)

↑ reference

→ this is the address immediately after the last item

* list.pop() removes & return.

x = list.pop() → last items gets removed

id(list) + 24 + 8 + (8 * (length - 1))

Date : .....
Page : .....

list.insert(index, value) · → Inserts goes in the list & every element items get dismantle.

If I do insert:
list.~~index~~insert(0, Sap)

length = 6

index	value
0	Alice
1	Bob
2	John
3	Nop
4	roar
5	100

Sap → shift

How many moves: 6 so, O(n)

In case of large dataset, shifting element is not a good idea - so,,

pop() When you pop 0th idex element, every element items needs to be shifted left side so, O(n)

If this Poped: ← 0 :

1 Alice
2 Bob
3 John
4 12345
5 16

List.remove('Alice') → Search + remove & don't retu
O(n) + O(n-1) - Worst case

remove: Worst case ⇒ O(n)

Search: Combn of search & remove becomes O(n).

append(): If memory full. It takes the list & find a newspace