List ADT

IT5003: Data Structures and Algorithms
(**AY2019/20 Semester 1**)

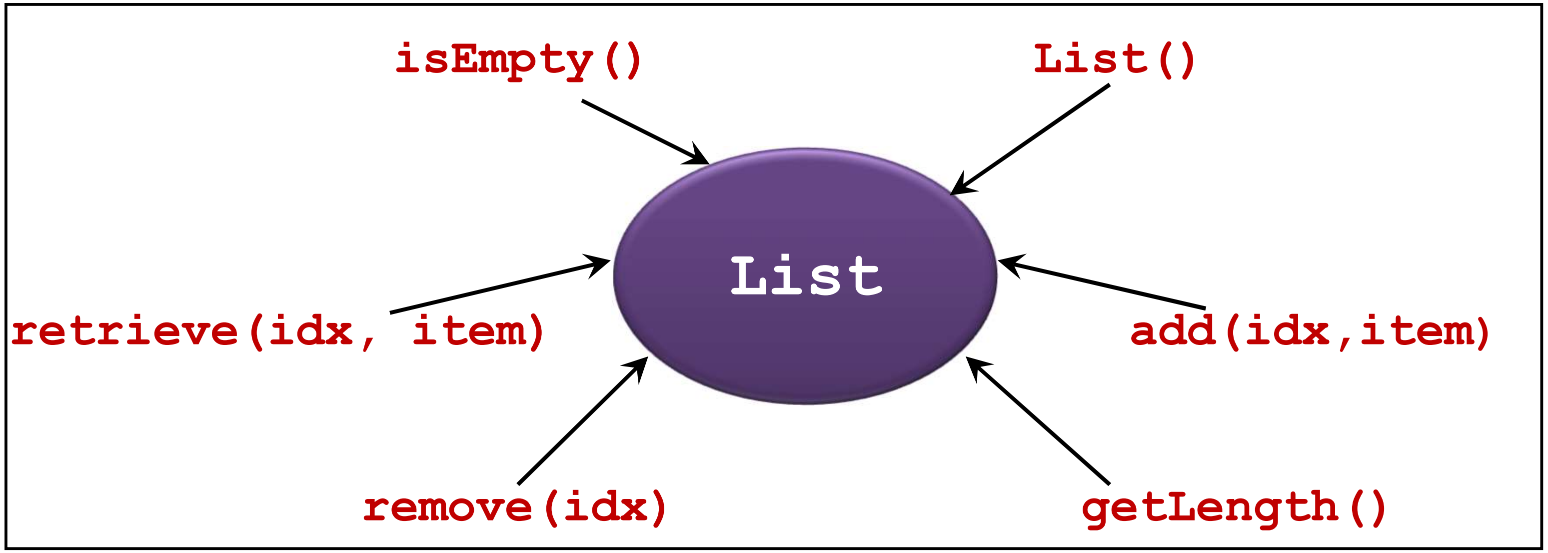
Lecture Overview

- List ADT
  - Specification

- Implementation for List ADT
  - Array Based
    - Pros and Cons
  - Linked List Based
    - Pros and Cons

List ADT

- A sequence of items where positional order matter $<a_1,\ a_2, \ldots,\ a_{n-1},\ a_n>$
- Lists are very pervasive in computing
  - e.g. student list, list of events, list of appointments etc



The **List** ADT

idx : Position, integer
item : Data stored in list, can be any data type

List ADT : **Python Specification**

```python
#imports not shown
class ListBase(ABC):
    @abstractmethod
    def isEmpty(self):
        pass

    @abstractmethod
    def getLength(self):
        pass
```

Operations to check on the state of list.

```python
    @abstractmethod
    def insert(self, index, newItem):
        pass

    @abstractmethod
    def remove(self, index):
        pass

    @abstractmethod
    def retrieve(self, index):
        pass

    @abstractmethod
    def toString(self):
        pass
```

The three major operations

Operation to ease printing & debugging.

ListBase.py

Two Major Implementations

1. Array implementation
2. Linked list implementation

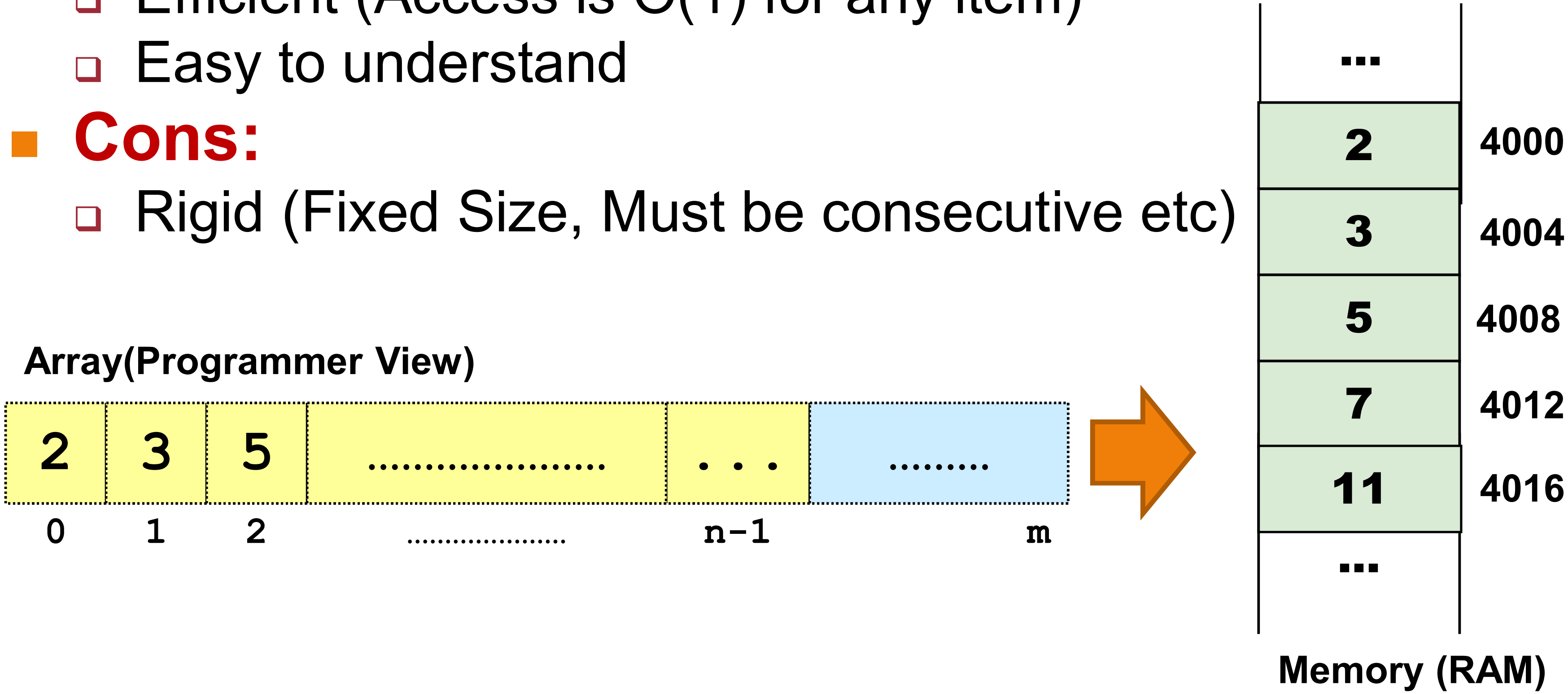
- General steps:
  1. Choose an **internal data structure**
     - e.g. Array or linked list
  2. Figure out the algorithm needed for each of the major operations in List ADT:
     - **insert**, **remove** and **retrieve**
  3. Implement the algorithm from step (2)

Consecutive Memory Locations

LIST ADT USING ARRAY

Array: **A Quick Overview**

- Array is commonly the simplest data structure in a programming language
  - Simply expose the actual memory layout to coder
- **Pros:**
  - Efficient (Access is O(1) for any item)
  - Easy to understand
- **Cons:**
  - Rigid (Fixed Size, Must be consecutive etc)



Array in Python

- Interestingly, Python "hides" array by providing a "better" basic data structure: **List**
  - Essentially wraps the low level array with higher level functionality (an example of ADT!)
  - E.g. Dynamically expands, Allow different type of data in the collection, Checks for index etc.
- So…..
  - We have to work extra hard to get Python exposing the lower level data structure
  - Essentially, we are studying how the Python "List" is actually implemented in this section

Python: `ctypes` Library

- A library that exposes low level functionality for Python
  - Also allow cross-operation with **C programming language**

```python
from ctypes import *

def main():
    array = (5 * c_int)()

    for i in range (5):
        array[i] = 3000 + i

    for i in range (5):
        print("Item[%d] = %d" %(i, array[i]))

    print(hex(addressof(array)))

    #cause exception
    array[10] = 123
```

Import library

`array` is a fixed size container with 5 integers

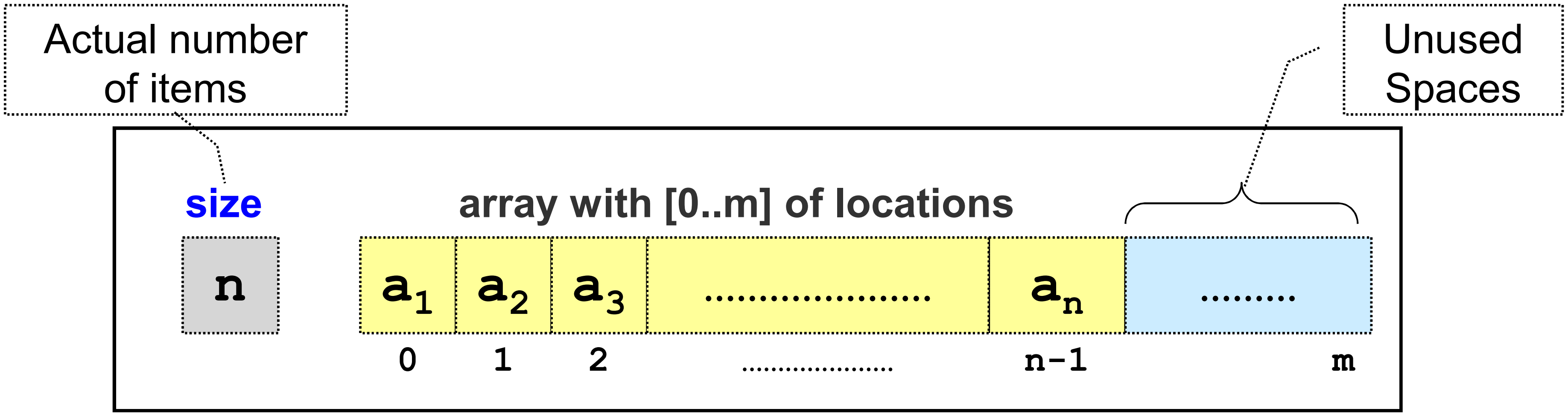
`array` does not dynamically expand

```
Item[0] = 3000
Item[1] = 3001
Item[2] = 3002
Item[3] = 3003
Item[4] = 3004
0x26fef81b258
...<Trace Back Omitted>...
IndexError: invalid index
```

`Demo-ctypes.py`

Implement List ADT: Using Array

- Array is a prime candidate for implementing the ADT
  - Simple construct to handle a collection of items
- **Advantage:**
  - Very fast retrieval

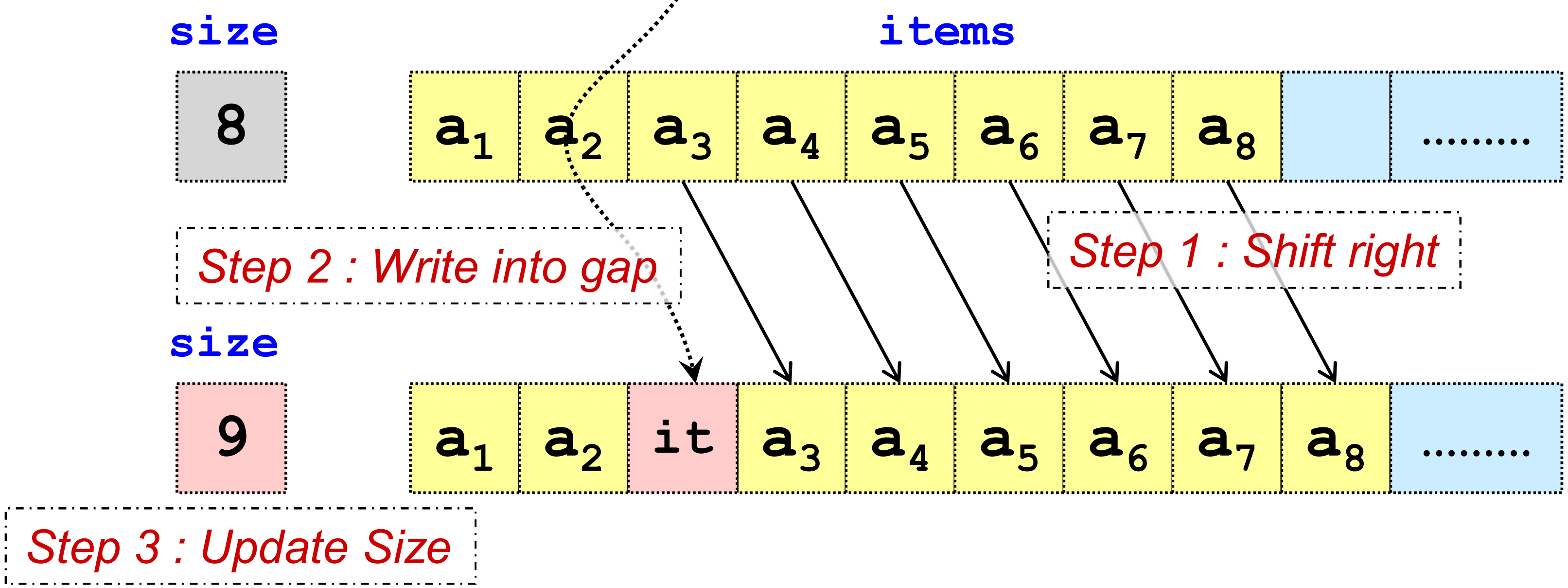


Internal of the `List` ADT, Array Version

Insertion : Using Array

- **Simplest Case:** Insert to the end of array
- Other Insertions:
  - Some items in the list needs to be shifted
  - **Worst case:** Inserting at the head of array

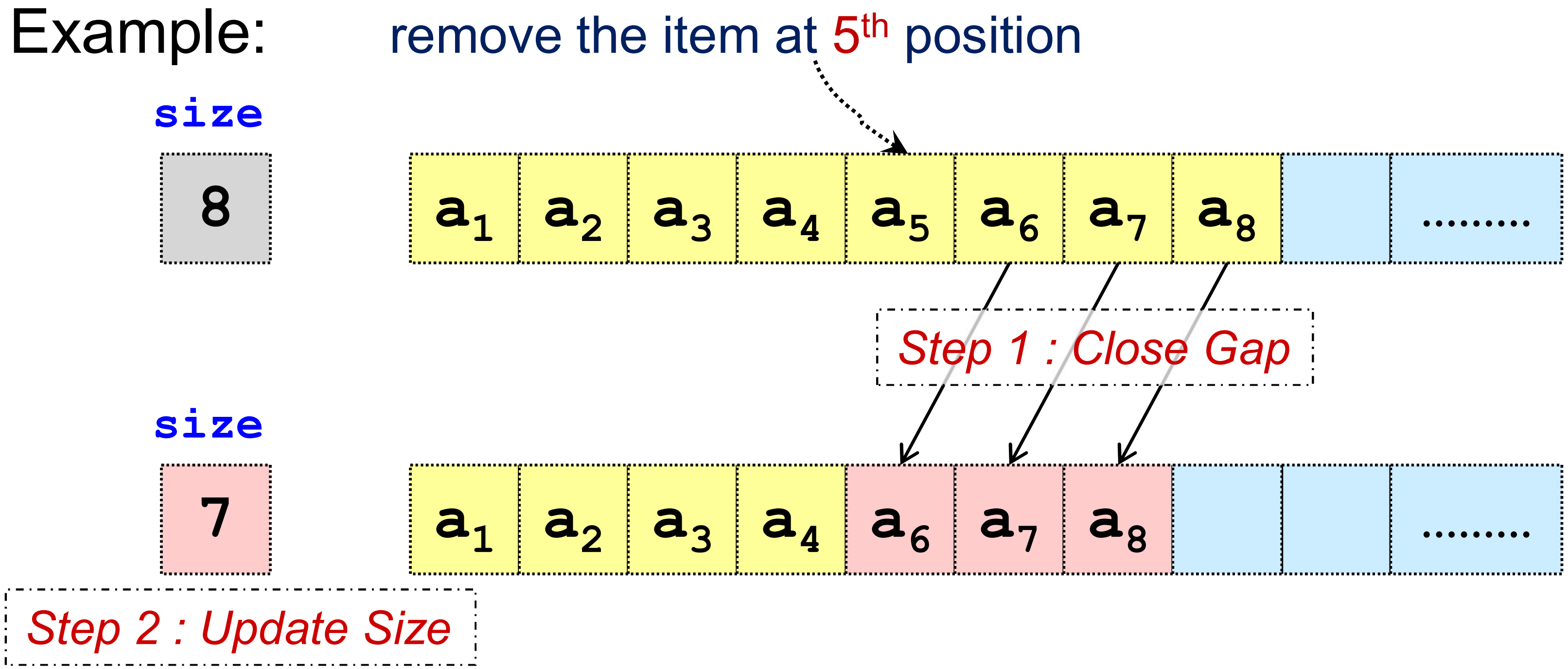
Example : Insert item "*it*" into the 3rd position



Deletion: Using Array

- **Simplest Case:** Delete item from the end of array
- Other deletions:
  - Items needs to be shifted
  - **Worst Case:** Deleting at the head of array

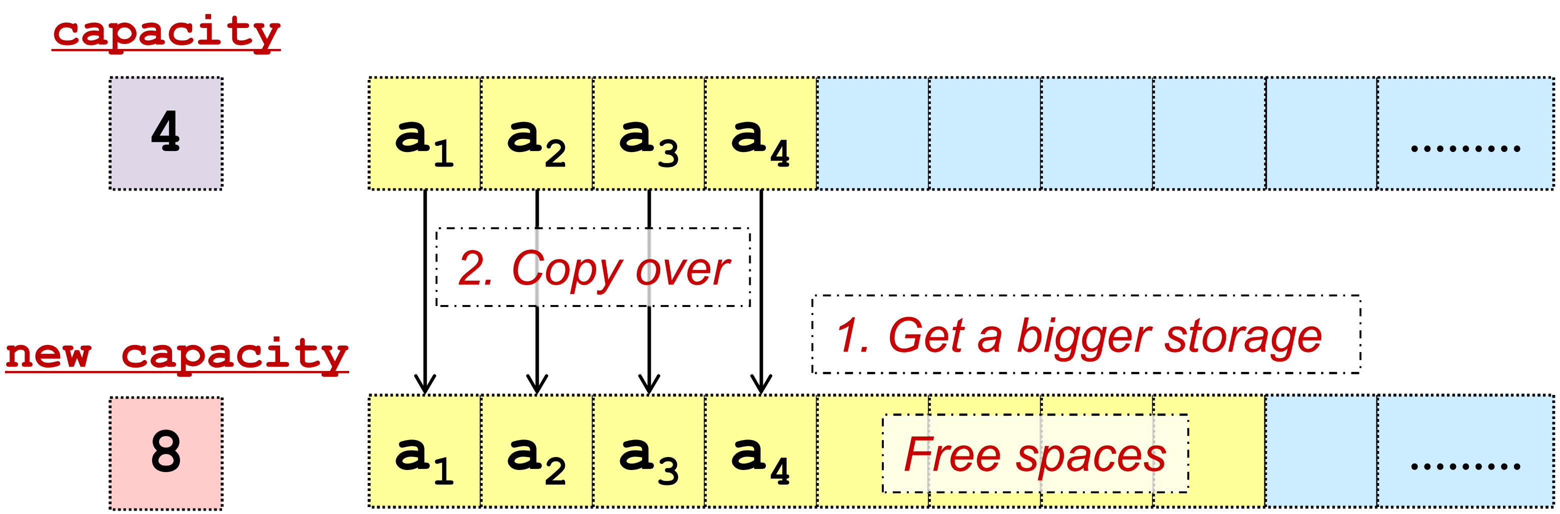
Example:



Resizing: Using Array

- Using a fixed size storage ➔ Can run out of space
- Common solution:
  - Get a larger piece of storage (How much larger?)
  - Move the data from old data over

Example: Original storage is size 4 and already full



ListArray: Implementation (1/4)

```
class ListArray(ListBase):

    def __init__(self):
        self._size = 0
        self._enlarge(1)

    def _enlarge(self, newCapacity):
        newStorage = (newCapacity * ctypes.py_object)()
        for i in range (self._size):
            newStorage[i] = self._storage[i]
        self._storage = newStorage
        self._capacity = newCapacity

    def isEmpty(self):
        return self._size == 0

    def getLength(self):
        return self._size
```

The storage is a fixed size stretch of consecutive locations

Resizing the internal storage when needed

ListArray.py

ListArray: Implementation (2/4)

```
def insert(self, index, newItem):
    if index < 1 or index > self._size+1:
        return False
    if self._size == self._capacity:
        self._enlarge(self._capacity*2) # grow 2x in capacity

    internal = index - 1 #internal index in [0..size-1]
    for pos in range(self._size-1, internal-1,-1 ):
        self._storage[pos+1] = self._storage[pos]

        self._storage[internal] = newItem
        self._size += 1
```

The Insertion Steps as discussed

ListArray: Implementation (3/4)

The Deletion Steps as discussed

```
def remove(self, index):
    if index < 1 or index > self._size:
        return False
    internal = index - 1 #internal index in [0..size-1]

    for pos in range(internal, self._size-1):
        self._storage[pos] = self._storage[pos+1]
    self._size -= 1
```

ListArray: Implementation (4/4)

Access is very efficient!

```
def retrieve(self, index):
    if index < 1 or index > self._size:
        return None
    internal = index - 1 #internal index in [0..size-1]

    return self._storage[internal]
```

A simple display to show all items in the List

```
def toString(self):
    str = "Size[{:d}/{:d}] | ".format(self._size,
    self._capacity)
    for i in range(self._size):
        str += "[{}]".format(self._storage[i])
    return str
```

Using List ADT: User Program

- Instead of an actual List ADT application:
  - We write a program to **test the implementation** of various List ADT operations

- Pay attention to **how we test** the operations:
  - For each operations:
    - Test different scenarios, basically to exercise different "decision path" in the implementation
  - For example, to test the **insert** operation:
    - Insert into an empty list
    - Insert at the first, middle and last position of the list
    - Insert with incorrect index

List ADT : Sample User Program 1/2

```
def main():
    l = ListArray()

    if l.insert( 1, 333 ):
        print("333 Insertion successful!\n")
    else:
        print( "333 Insertion FAILED!\n" )
    print(l.toString())

    l.insert( 1, 111 )
    l.insert( 3, 777 )
    l.insert( 3, 555 )
    print("After a few insertions: "+l.toString())


    print("First item is %d" % l.retrieve(1))
    print("Last item is %d" % l.retrieve(l.getLength()))
```

Using the array implementation of list

If the insertion is implemented properly, the list should contain **[111 333 555 777]** at this point

Test **retrieve**() and **getLength**()

User-List-Testing.py

List ADT : Sample User Program 2/2

```
l.remove(1)
l.remove(2)
l.remove( l.getLength() )

print("After a few deletions:")
print("First item is %d" % l.retrieve(1))
print("Last item is %d" % l.retrieve(l.getLength()))
print("Final List is"+l.toString())
```

Test **`remove`**():

-remove 1st item
-remove last item
-remove item in the middle

- We will reuse this program to test other List ADT implementation later

Array Implementation: **E**fficiency (**T**ime)

Retrieval:

- **Fast:** one access

Insertion:

- **Best case:** No shifting of elements
- **Worst case:** Shifting of all **N** elements

Deletion:

- **Best case:** No shifting of elements
- **Worst case:** Shifting of all **N** elements

Array Implementation : Efficiency (Space)

- Size of array is restricted to **`Capacity`** at any point in time

- **Problem:**
  - The "right" capacity is **not known (or cannot be known) in advance**
    - **`Capacity`** is too big == unused space is wasted
    - **`Capacity`** is too small == run out of space easily
      ➔ Need to copy ➔ Impose Copy Overhead

Array Implementation : Observations

- For **fixed-size collections**
  - Arrays are **great**

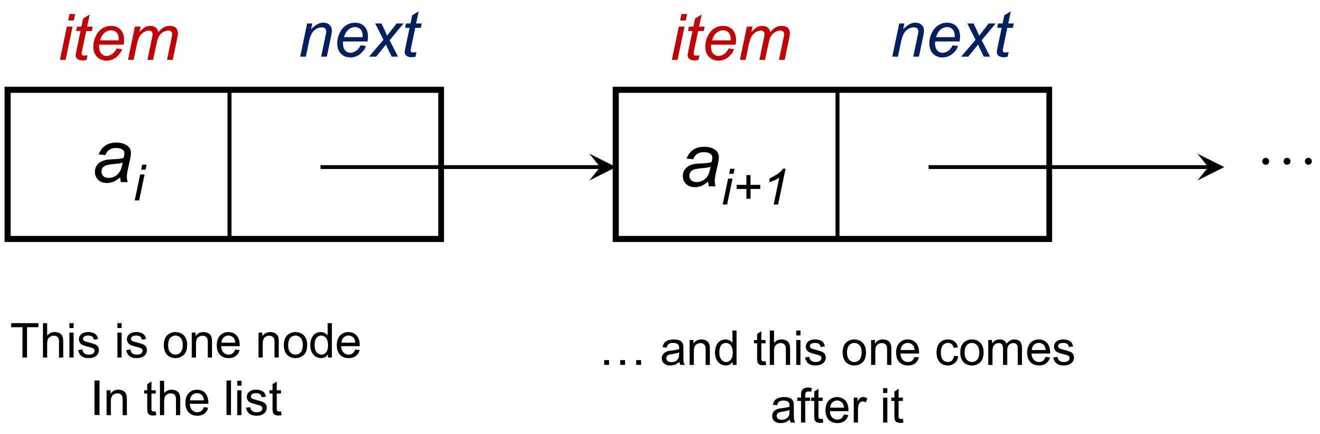
- For **variable-size collections**, where dynamic operations such as insert/delete are common
  - Array is a **poor choice** of data structure.

  - For such applications, ***there is a better way……***

LIST ADT USING LINKED LIST

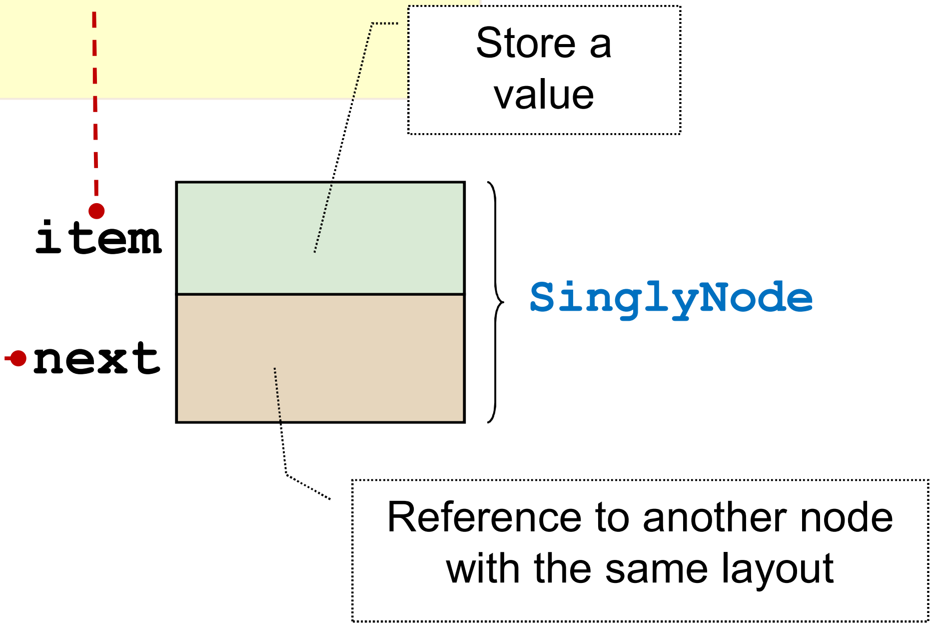
Implement List ADT: Using Linked List

- Reference(Pointer) Based Linked List:
  - Allow elements to be **non-contiguous** in memory
  - Order the elements by associating each with its **neighbour(s)** through reference



Linked List: One Node

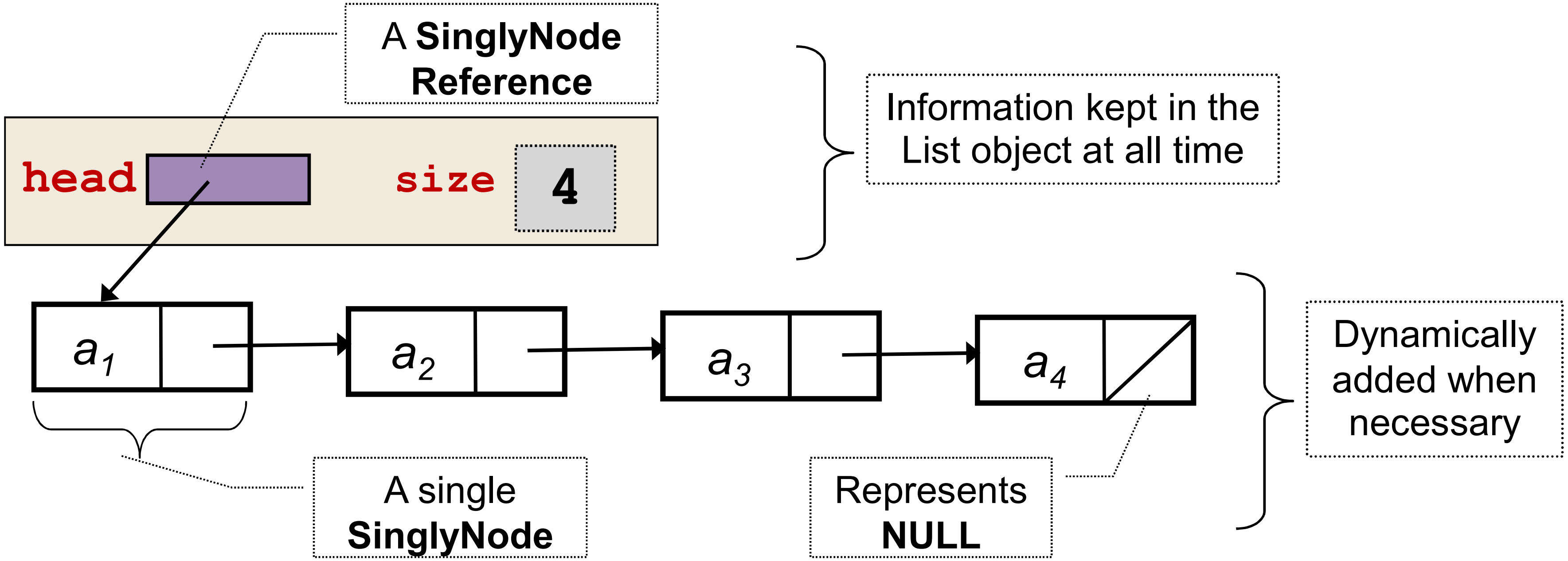
```
class SinglyNode:
    def __init__(self, item, link = None):
        self.item = item
        self.next = link
```



- "Singly" means there is only a **Single** reference to the next node

List ADT: Using Linked List

- List of four items $< a_1, a_2, a_3, a_4 >$



- We need:
  - **head** reference to indicate the first node
    - Other nodes are accessed by "hopping" through the **next** reference
  - **size** for the number of items in the linked list

Linked List Implementation: Design

- Linked list implementation is more complicated:
  - Need to handle a number of scenarios separately

- Let us walkthrough the **insertion** algorithm in detail:
  - Highlight the importance of design before coding
  - Highlight the design considerations:
    - How to consider different cases?
    - How to modularize and reuse common code?

Linked List Insertion: **General**

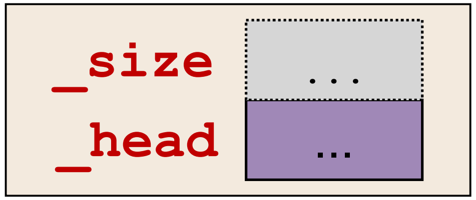
- List ADT provides the `insert()` method to add an item:
  - The new item itself is given
  - The index [**1**…**size + 1**] of the new item is given

- Due to the nature of linked list, there are several possible scenarios:
  1. Item is added to an **empty** linked list
  2. Item is added to the **head (first item)** of the linked list
  3. Item is added to the **last position** of the linked list
  4. Item is added to the **other positions** of the linked list

Linked List Insertion: **Preliminary**

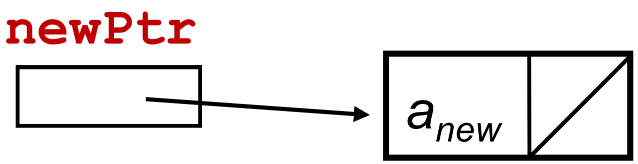
- The **`ListLinkedList`** object stores:
  - Head reference and the current size of linked list

```
class ListLinkedList(ListBase):
    def __init__(self):
        self._head = None
        self._size = 0
  ...
```

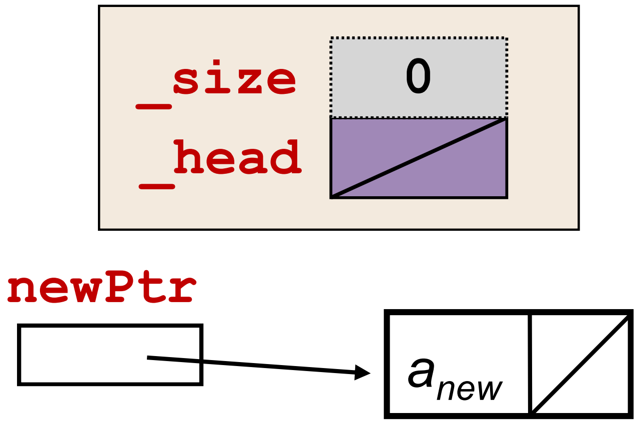


- We need to ***construct*** a new linked list node to store the new item

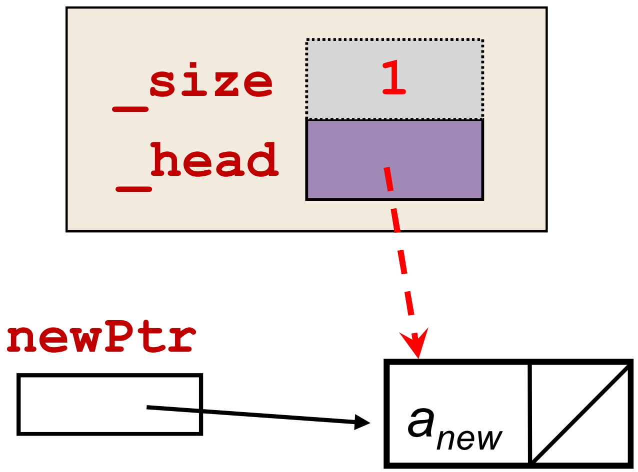
```
newPtr = SinglyNode(a_new)
```



Insertion Case 1: Empty Linked List

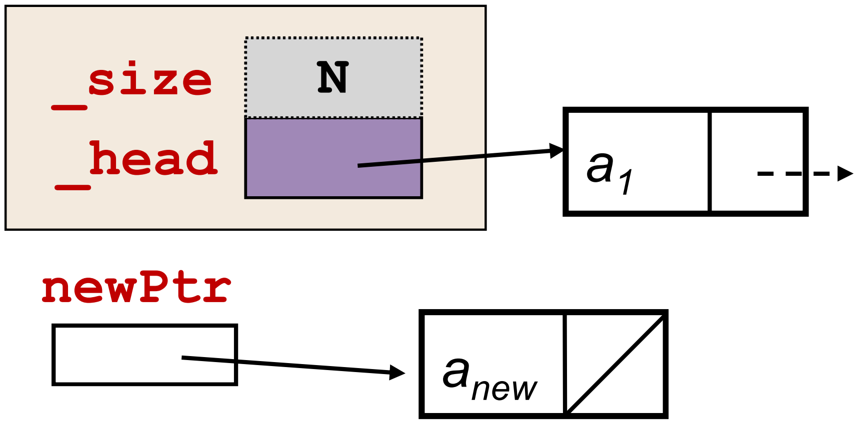



```
self._size += 1

self._head = newPtr
```

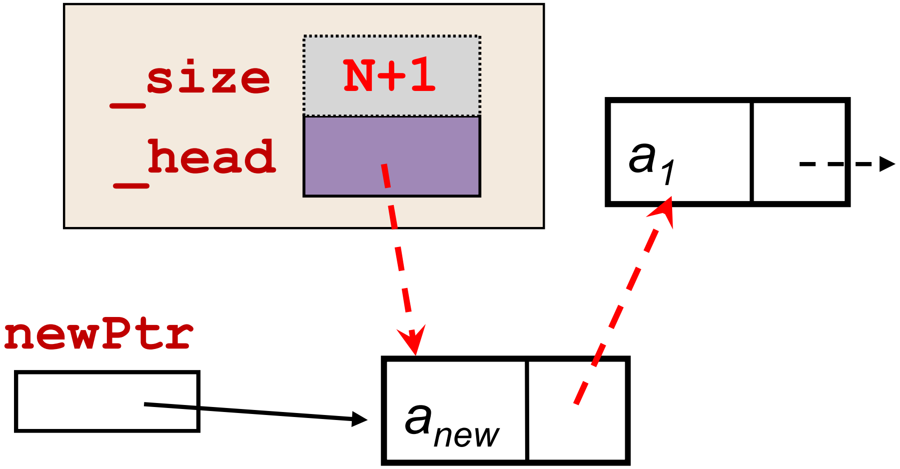


Question: is **`newPtr`** needed after this operation?

Insertion Case 2: **Head of Linked List**



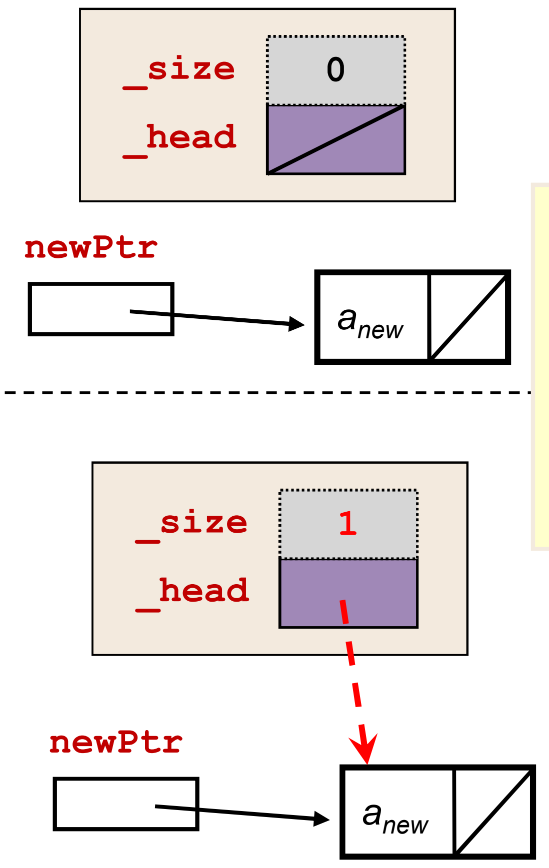

```
self._size += 1

newPtr.next = self._head
self._head = newPtr
```



Question: Very similar to previous case, can we combine?

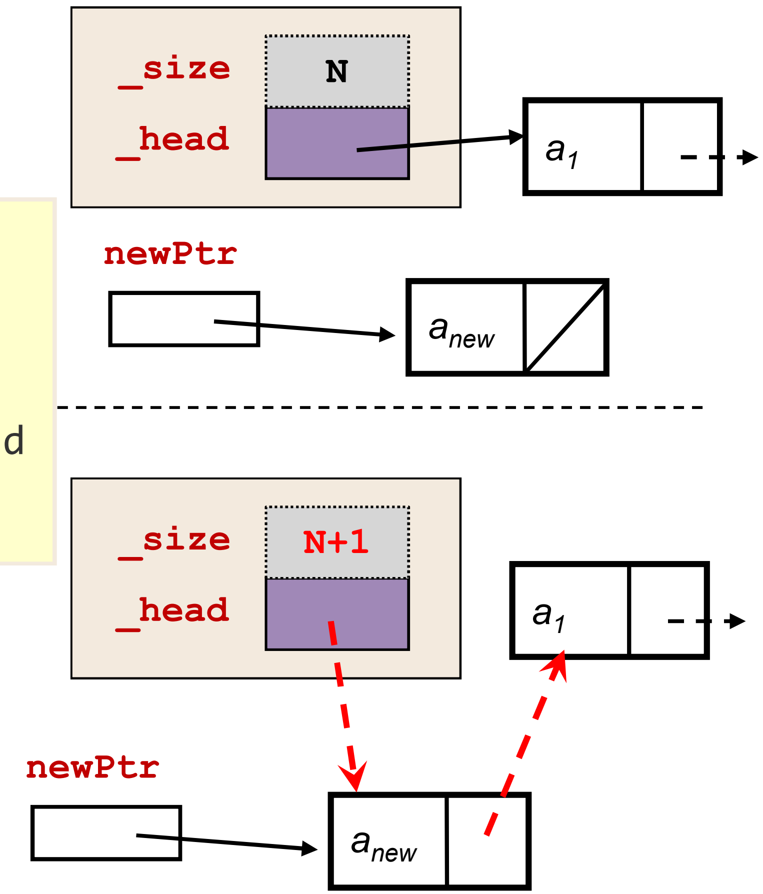
Insertion Case 1 and 2: **Common Code**

- Insert into head of linked list (possibly empty)



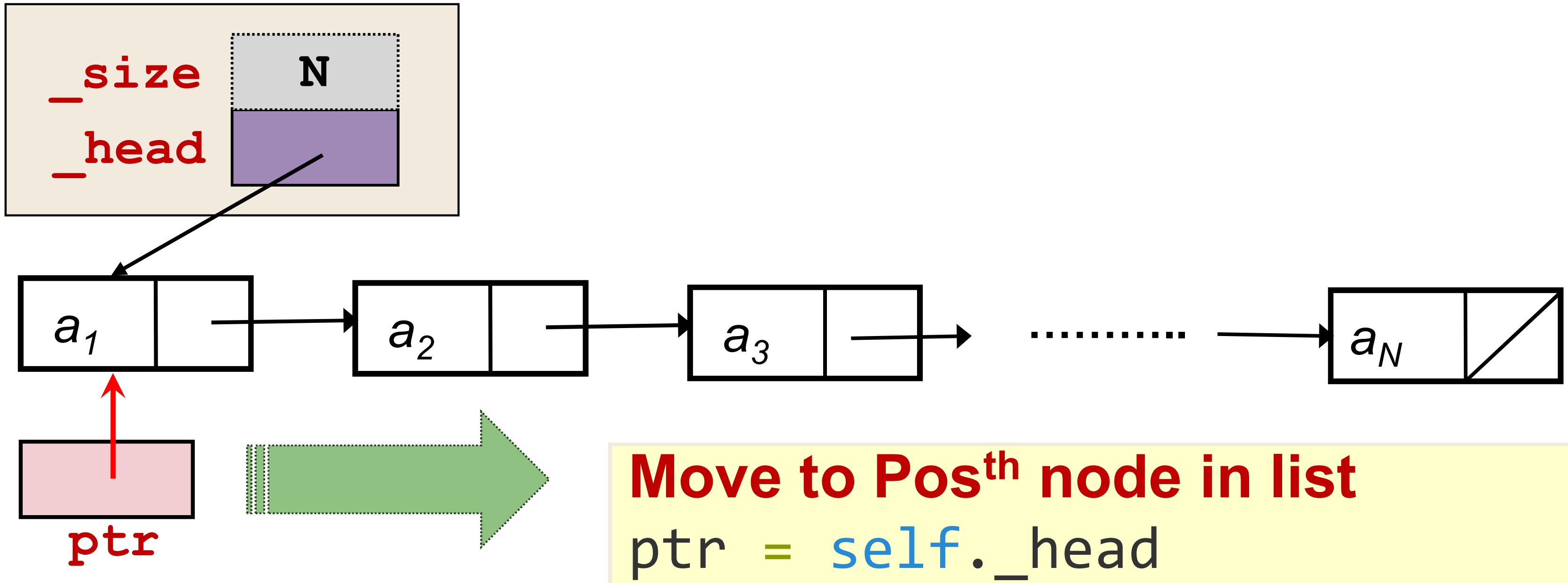
Can use the same code:

```
self._size += 1

newPtr.next = self._head
self._head = newPtr
```



Linked List Insertion: Traversal

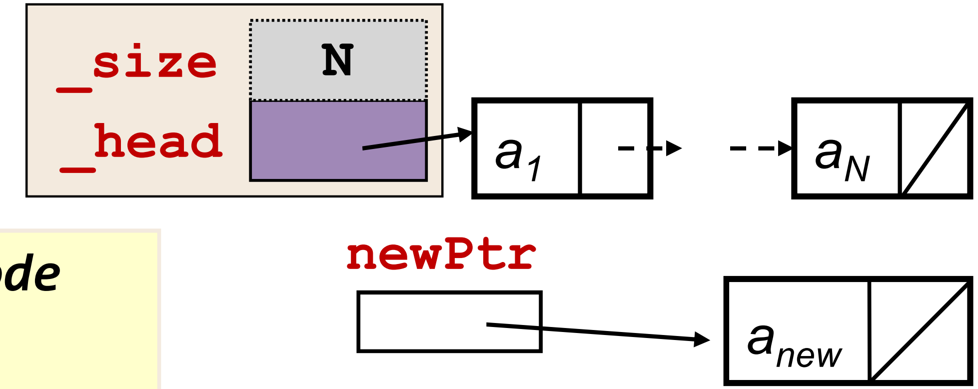
- Since we only keep the head pointer, **list traversal** is needed to reach other positions
  - Needed for insertion case 3 and 4



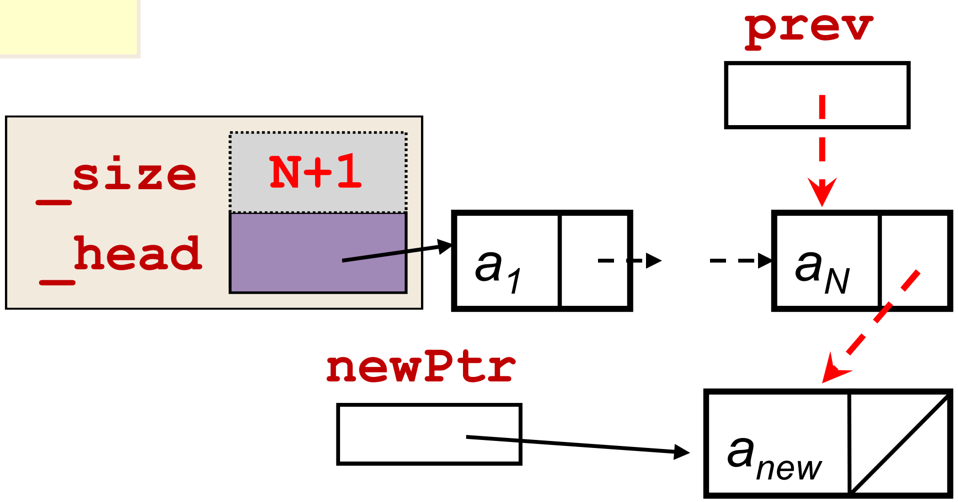
Move to Posth node in list

```
ptr = self._head

for i in range(1, index):
    ptr = ptr.next
```

Insertion Case 3: End of Linked List

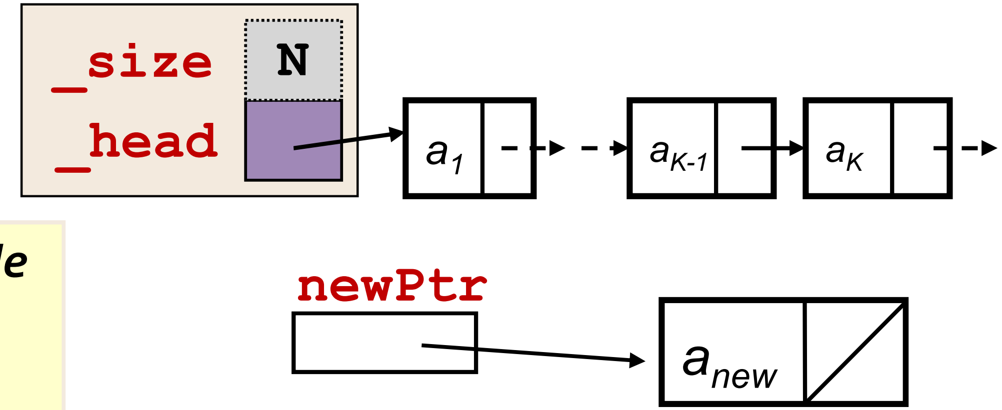



```
prev = Traverse to Nth node

newPtr.next = prev.next
prev.next = newPtr

self._size += 1
```

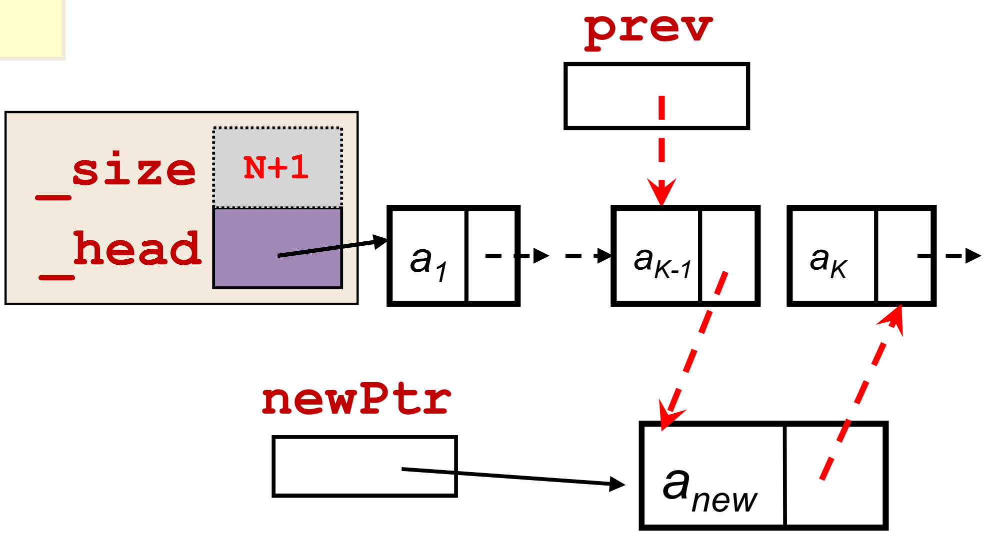


Use the list traversal code to move `prev` reference

Insertion Case 4: Kth Position (Middle)




```
prev = Traverse to K-1th node

newPtr.next = prev.next
prev.next = newPtr

self._size += 1
```

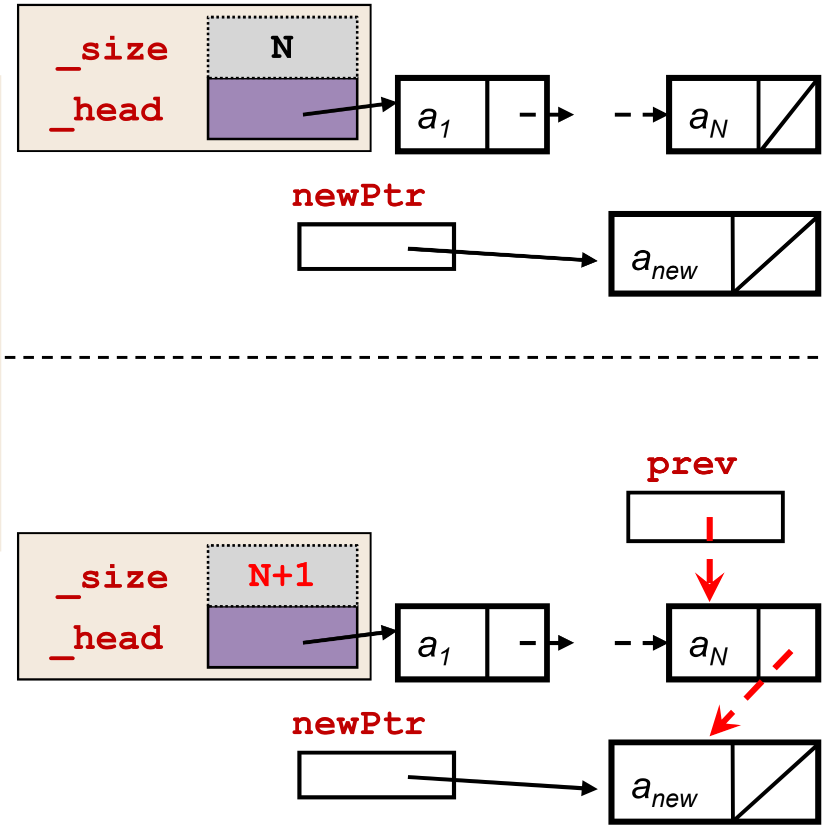


We only need to change at most two pointers for ANY insertion

Insertion Case 3 and 4: **Common Code**

- The code for case 4 happens to be a more general form of case 3:
  - Case 3 can be handled with the same code in case 4

```
K = N+1
prev = Traverse to K-1th node

newPtr.next = prev.next
prev.next = newPtr

self._size += 1
```



Insertion Code: Summary

- To insert **ItemNew** into **Index**th position
  - Assume **Index** is in range **[1….size + 1]**

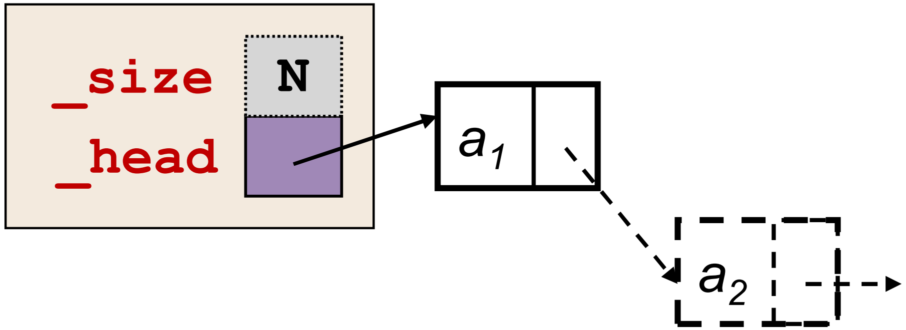
```
1. newPtr = a new SinglyNode
    i. item = itemNew
    ii.next reference = None

2. _size increases by 1

3. If Index is 1      //Case 1 + 2
    i. newPtr.next = _head
    ii._head = newPtr
4. Else               //Case 3 + 4
    i.  prev = Traverse to Index-1th node
    ii. newPtr.next = prev.next
    iii.prev.next = newPtr
```

Linked List Deletion: General

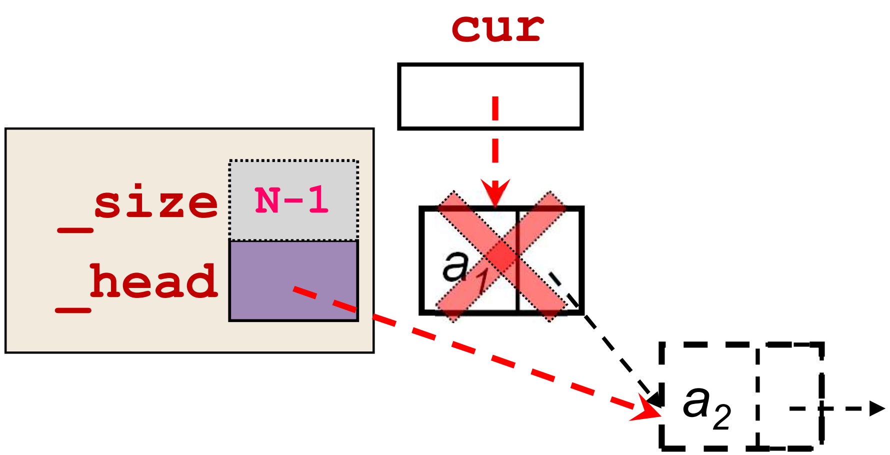
- For Linked List deletion, the cases can be simplified similar to:
  1. Deletion of head node (1st Node in list)
  2. Deletion of other node (including middle or end of list)

- Try to deduce the code logic using similar approach

Deletion Case 1: Head of Linked List



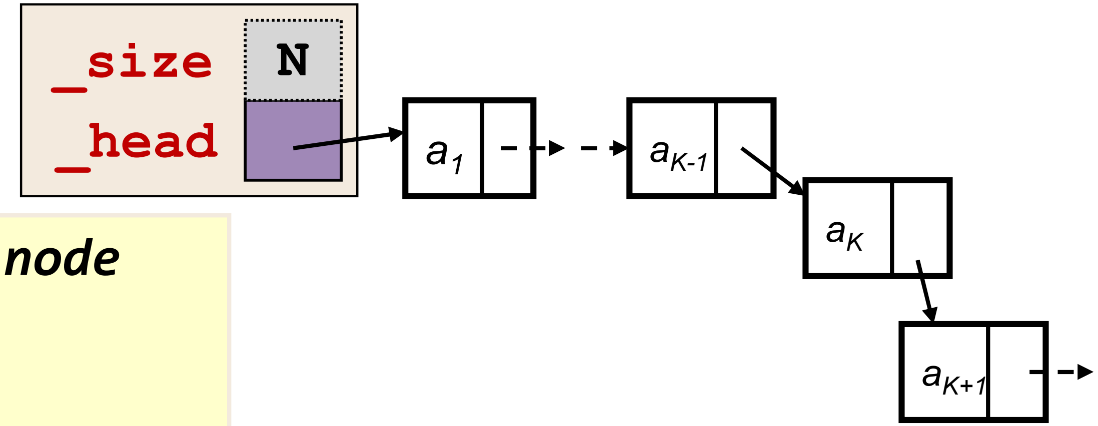

```
self._size -= 1

cur = self._head
self._head = self._head.n
ext
```



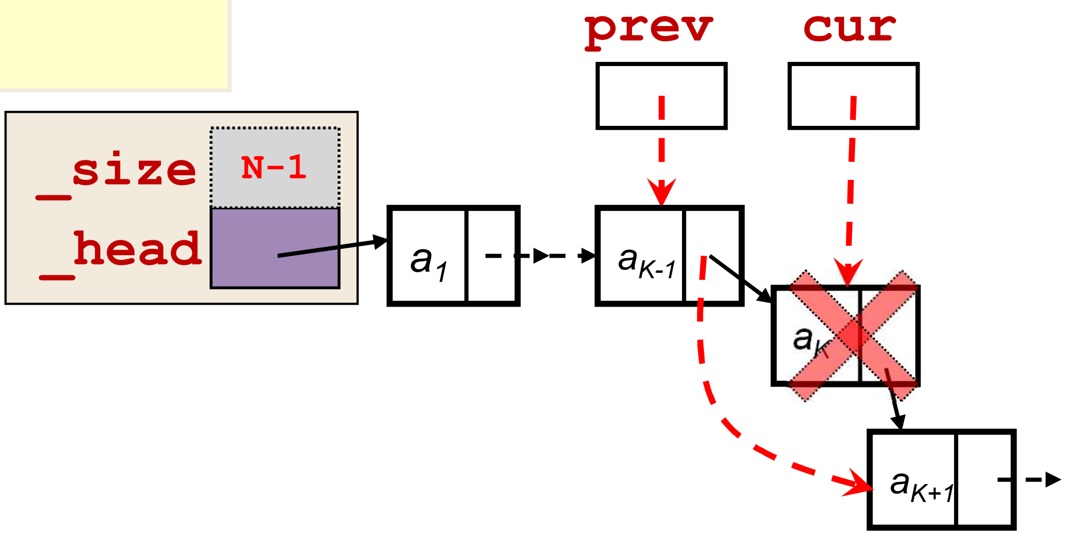
Question: What if there is only 1 node? Will the code work?

Question: How do we delete the cur node in Python?

Deletion Case 2: K^th Position (Middle)




```
prev = Traverse to K-1th node

self._size -= 1

cur = prev.next
prev.next = cur.next
```



Question: What if there is K^th node is the last node?

Deletion Code: Summary

- To delete item at `Index`th position
  - Assume `Index` is in range **[1….size]**

```
1. _size decreases by 1

2. If Index is 1      //Case 1
    i. cur = _head
    ii._head = _head.next
3. Else               //Case 2
    i.   prev = Traverse to Index-1th node
    ii.  cur = prev.next
    iii. prev.next = cur.next

4. Delete the cur node
```

The traversal code can be shared between insertion and deletion. Let's make it into another method

ListLinkedList: Implementation

```
class ListLinkedList(ListBase):

    def __init__(self):
        self._head = None
        self._size = 0


    def _traverse(self, index):
        if index < 1 or index > self._size:
            return None

        ptr = self._head
        for i in range(1, index):
            ptr = ptr.next
        return ptr
```

The **traversal** method is used internally only

ListLinkedList.py

ListLinkedList: Implementation

```
def insert(self, index, item):
    if index < 1 or index > self._size+1:
        return False

    newPtr = SinglyNode(item) #Create a new node
    if index == 1:
        newPtr.next = self._head
        self._head = newPtr
    else:
        prev = self._traverse(index-1)
        newPtr.next = prev.next
        prev.next = newPtr
    self._size += 1
    return True
```

Compare with the algorithm we figure out previously

`ListLinkedList`: Implementation

```
def remove(self, index):
    if index < 1 or index > self._size:
        return False

    if index == 1:
        cur = self._head
        self._head = self._head.next
    else:
        prev = self._traverse(index-1)
        cur = prev.next
        prev.next = cur.next
    self._size -= 1
```

Compare with the algorithm we figure out previously

`isEmpty()` and ***`getLentgh()`*** methods not shown as they have the same code

ListLinkedList: Implementation

```
def retrieve(self, index):

    if index < 1 or index > self._size:
        return None

    ptr = self._traverse(index)
    return ptr.item
```

What's the efficiency of retrieval?

- ***isEmpty()*** and ***getLentgh()*** methods not shown as they have the same code as before

List ADT : Sample User Program (Again!)

- With the Linked List implementation, we can test it using the same User Program from earlier

```
def main():
    l = ListLinkedList()

    if l.insert( 1, 333 ):
        print("333 Insertion successful!\n")
    else:
        print( "333 Insertion FAILED!\n" )

     //All other usage of List ADT remain
     // unchanged.
     .........
```

Using the linked list implementation of list

Linked List: Efficiency (Time)

Retrieval:

- ______________________________

Insertion:

- ______________________________

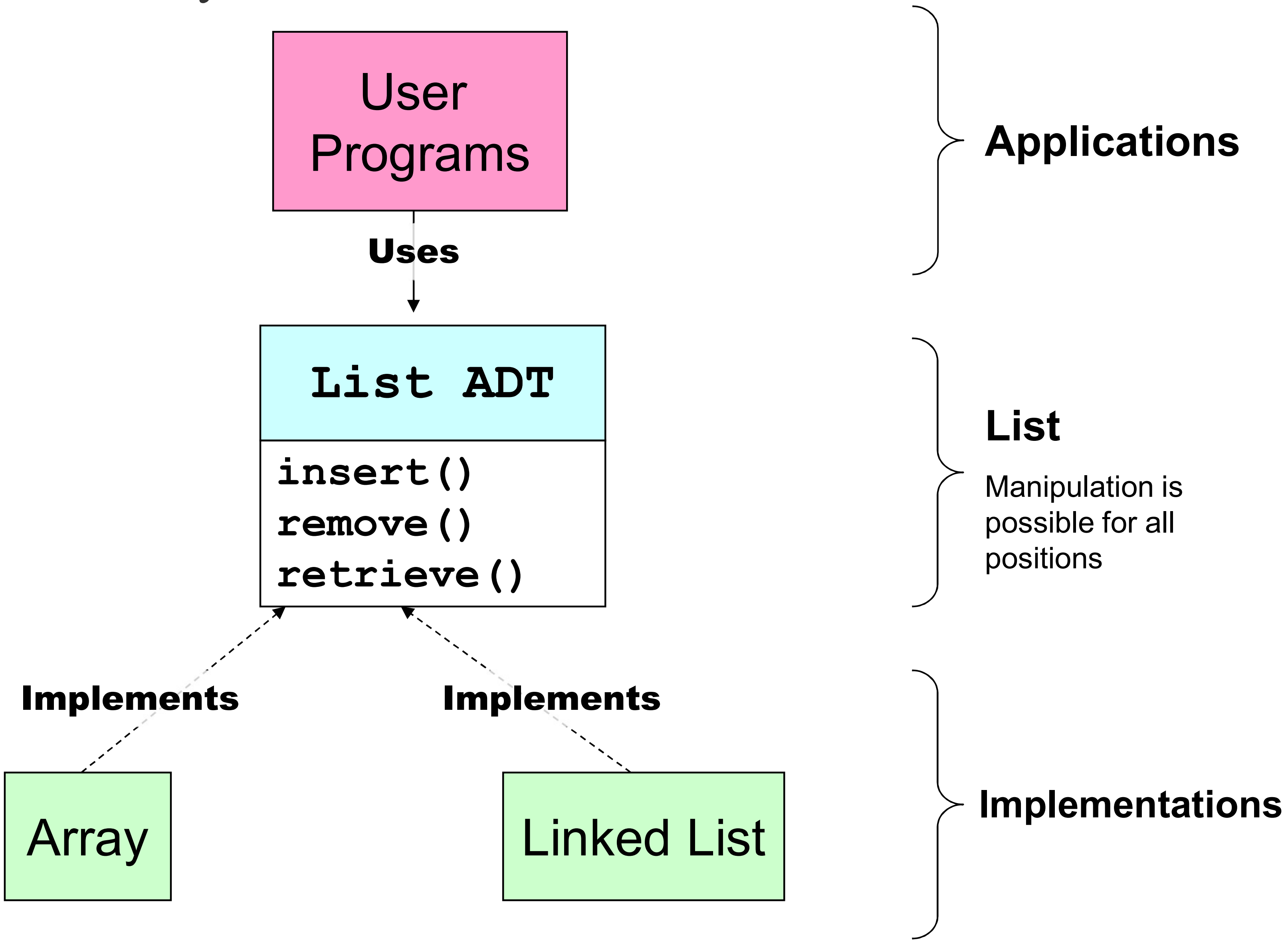
Deletion:

- ______________________________

Linked List: Efficiency (Space)

- Let's put what you've learned into practice
- Did Linked List solved the "capacity" issue we had with array implementation?
- Evaluate the space used for N items linked list in terms of Big-O notation

Summary


User Programs
Uses
List ADT
insert()
remove()
retrieve()
Implements
Implements
Array
Linked List
Applications
List
Manipulation is possible for all positions
Implementations

References

- [Carrano]
  - Chapter 3
    - List ADT and array based implementation
  - Chapter 4
    - Linked List and STL list
- [Koffman & Wolfgang]
  - Chapter 4.5 to 4.8

END