SINGLY

LINKED

LISTS

Common Collections in Programming

- Python lists are flexible
- can be used to implement other widely used data structures

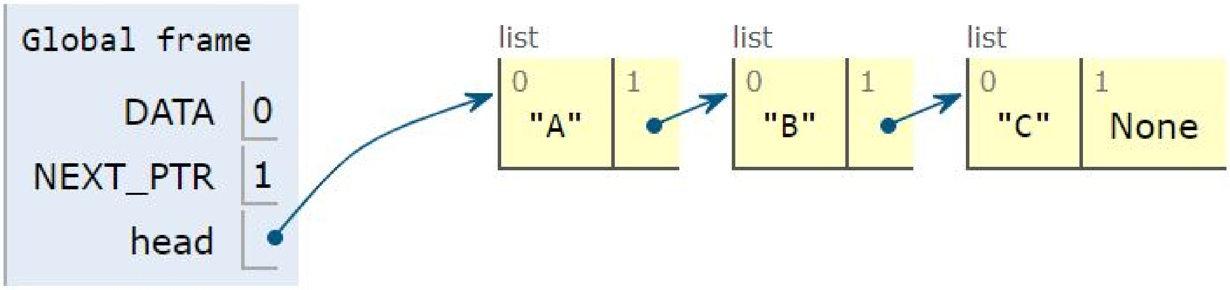
  1. singly linked lists
  2. doubly linked lists

Singly Linked List

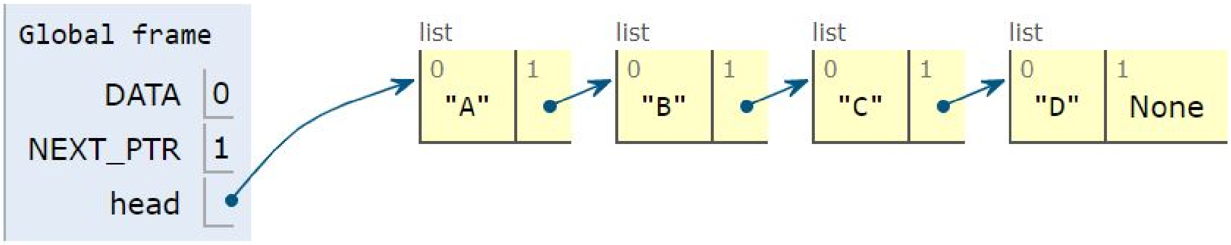
- linear collection of nodes
- each node contains:
  1. DATA field
  2. NEXT_PTR to the next node
- *head* points to start
- efficient insert/delete
- no random access

Example

- a single linked list with 3 nodes

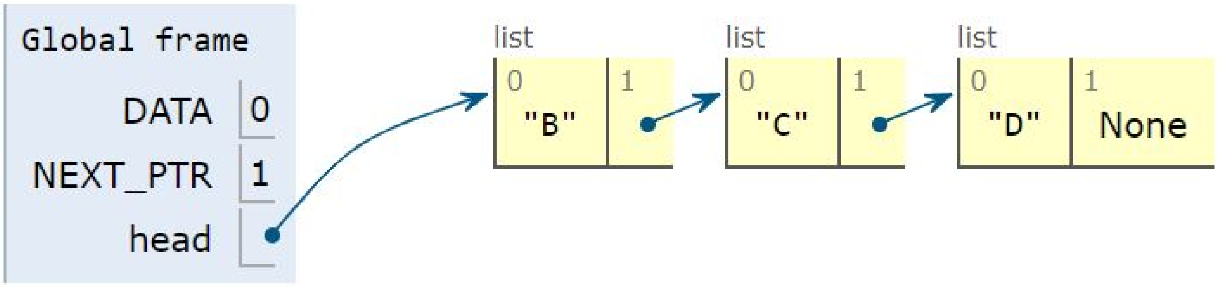


- insert new node at the end

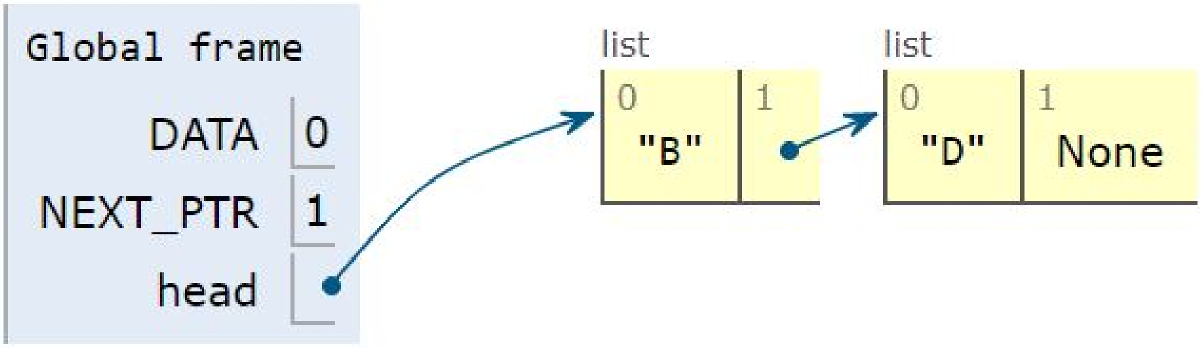


Example (cont'd)

- remove first node

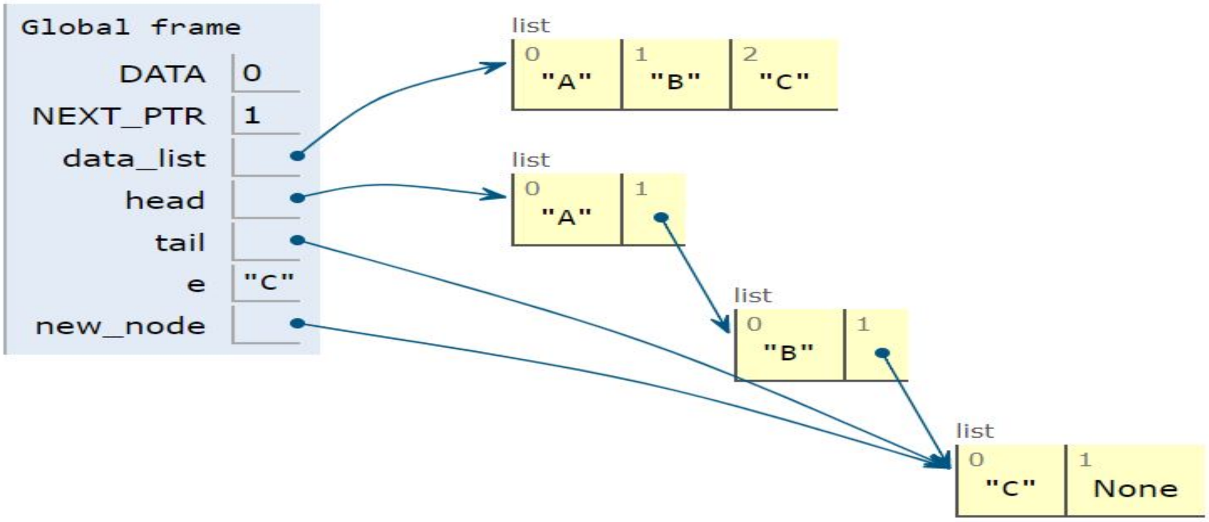


- remove intermediate node ”C”



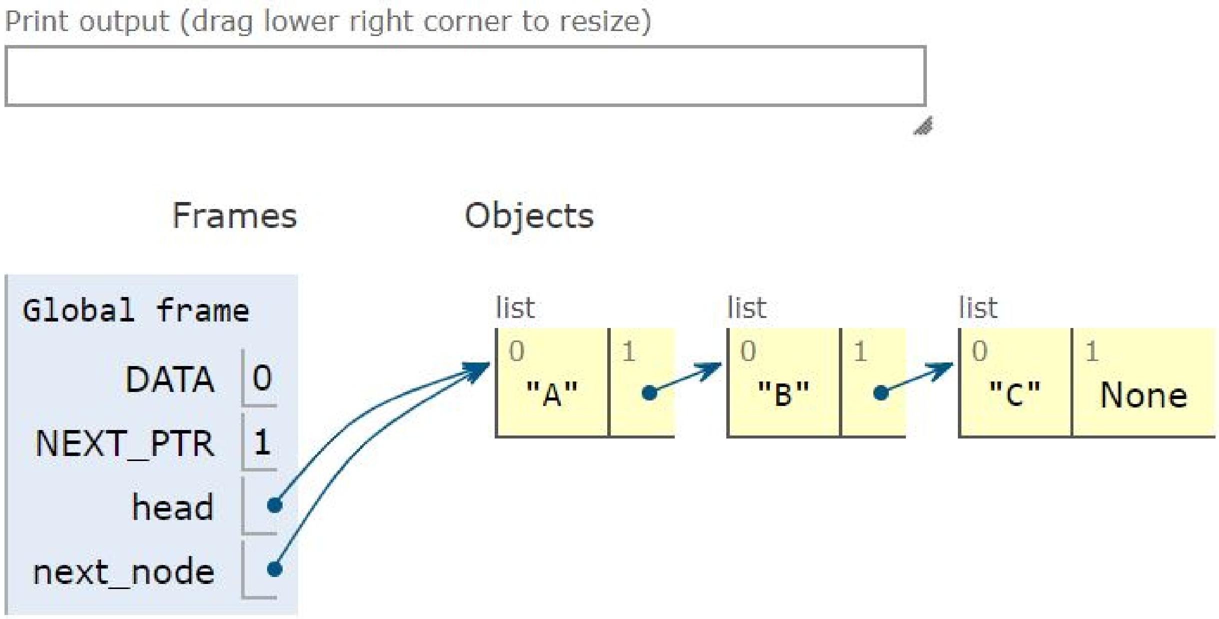
List Construction

```
DATA = 0;  NEXT_PTR = 1
data_list = [’A’, ’B’, ’C’]
head = None; tail = None

for e in data_list:
    new_node = [e, None]
    if head is None:
        head = new_node; tail = new_node
    else:
        tail[NEXT_PTR] = new_node;
        tail = new_node
```



List Traversal

```
next_node = head
while next_node is not None:
    print(next_node[DATA], end=' ')
    next_node = next_node[NEXT_PTR]
```



Global frame

DATA 0

NEXT_PTR 1

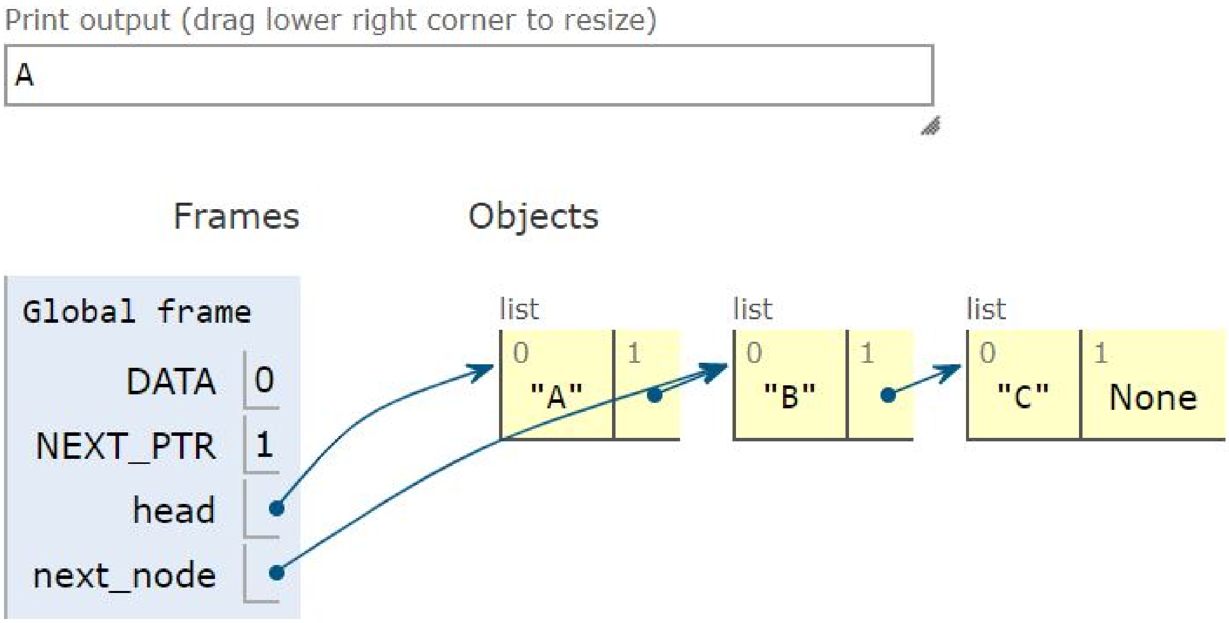
head

next_node

- initialization

List Traversal

```
next_node = head
while next_node is not None:
    print(next_node[DATA], end=’ ’)
    next_node = next_node[NEXT_PTR]
```



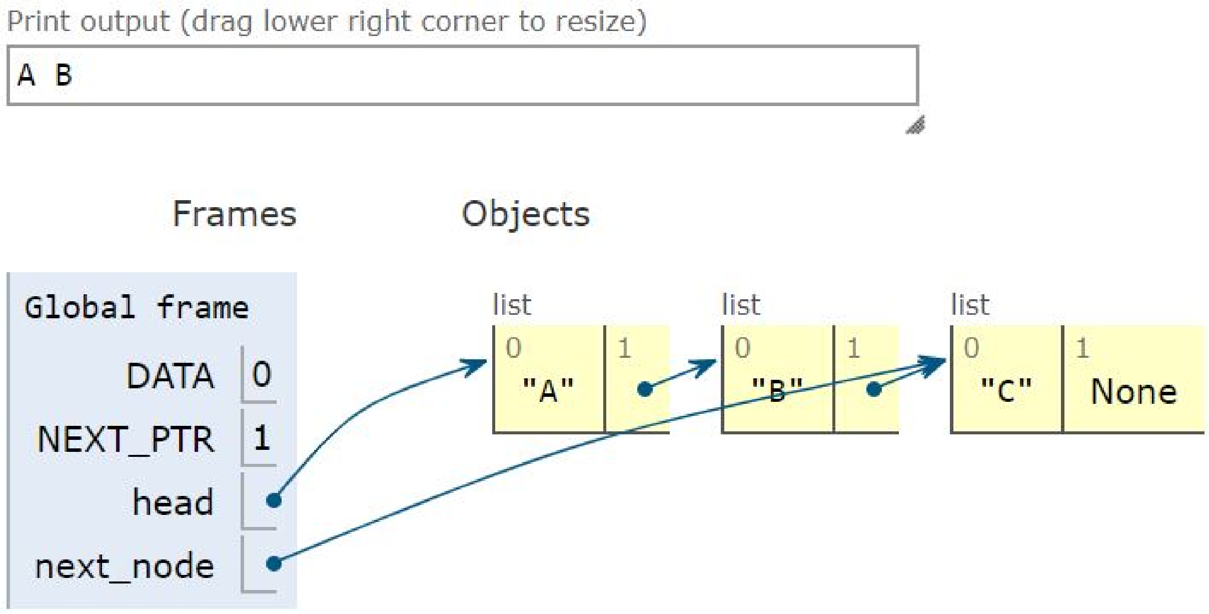
Print output (drag lower right corner to resize)

```
A
```

- after first node

List Traversal

```
next_node = head
while next_node is not None:
    print(next_node[DATA], end=' ')
    next_node = next_node[NEXT_PTR]
```

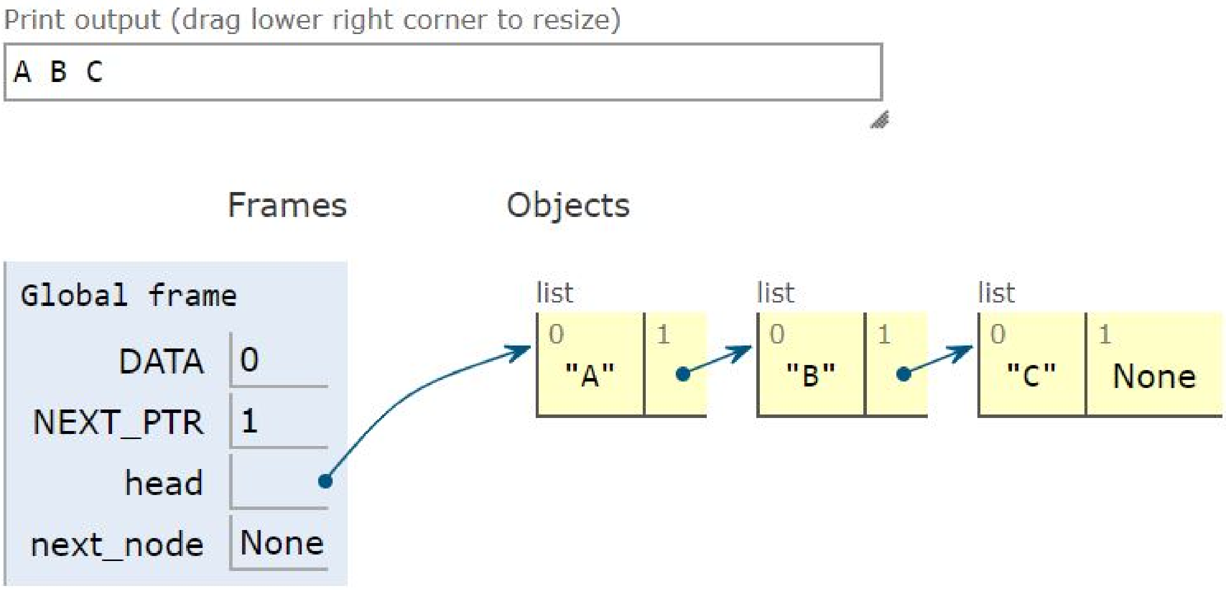


```
A B
```

- intermediate node(s)

List Traversal

```
next_node = head
while next_node is not None:
    print(next_node[DATA], end=' ')
    next_node = next_node[NEXT_PTR]
```



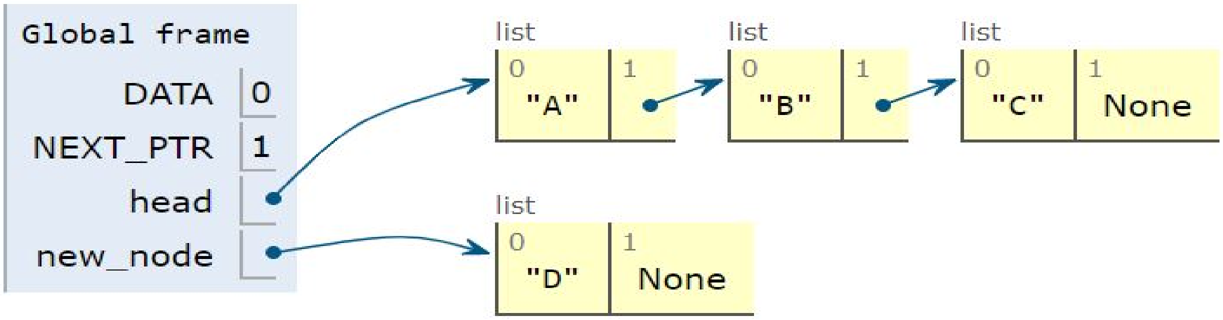
```
A B C
```

DATA	0
NEXT_PTR	1
head	
next_node	None

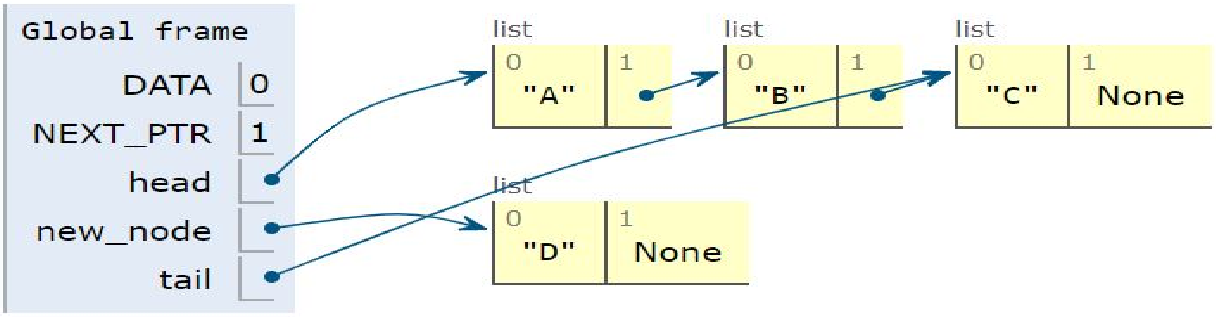
- after last node

Insert at List End

```
new_node = ['D', None]
```

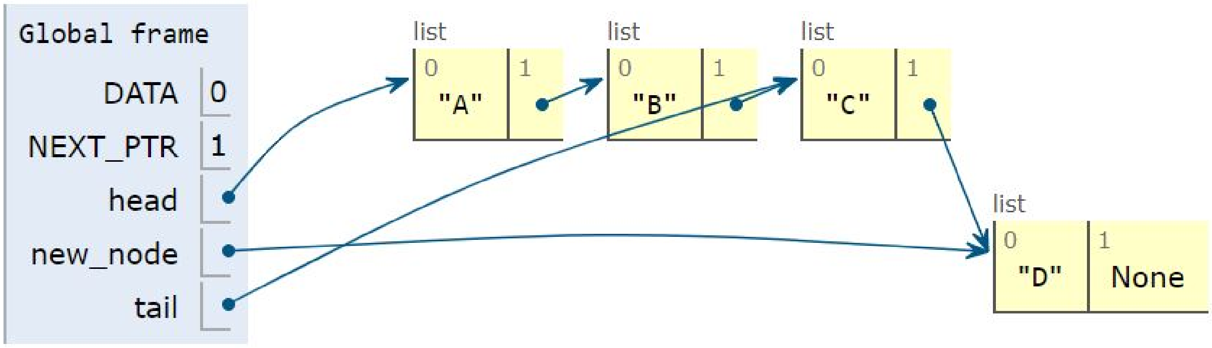


```
tail = head
while tail[NEXT_PTR] is not None:
    tail = tail[NEXT_PTR]
```



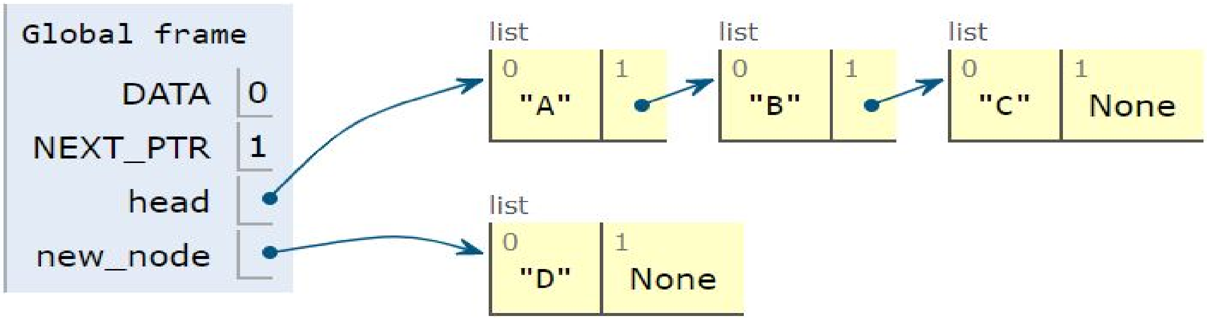
Insert at List End

```
tail[NEXT_PTR] = new_node
```

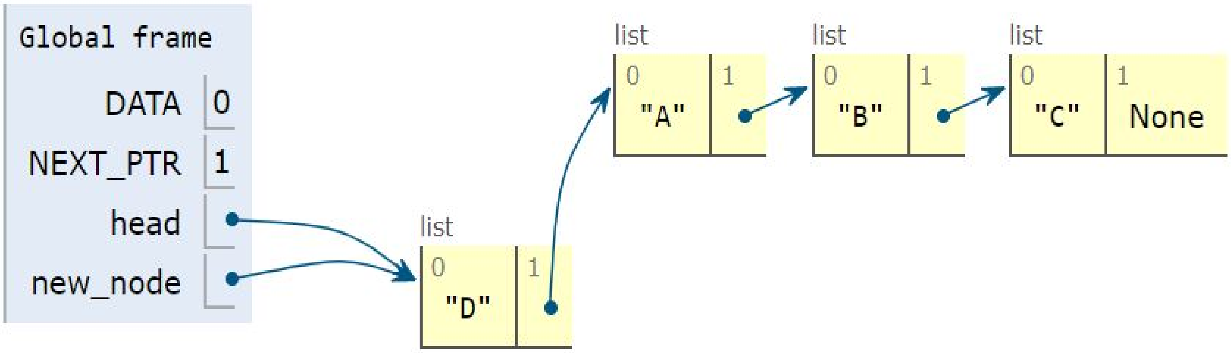


Insert at Head

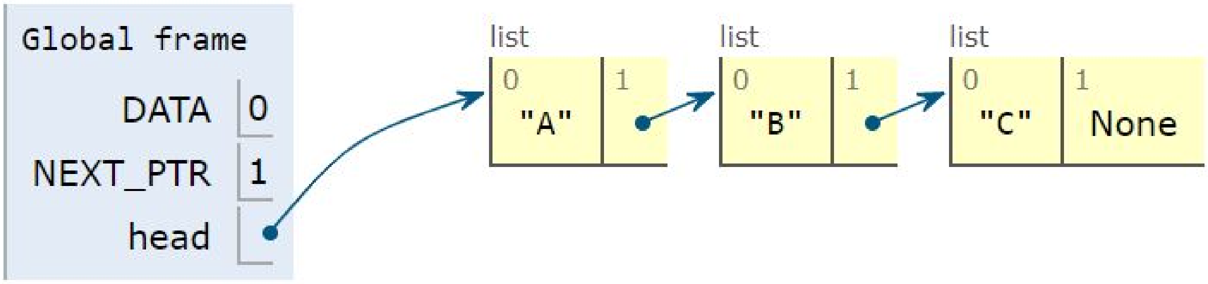
```
new_node = ['D', None]
```



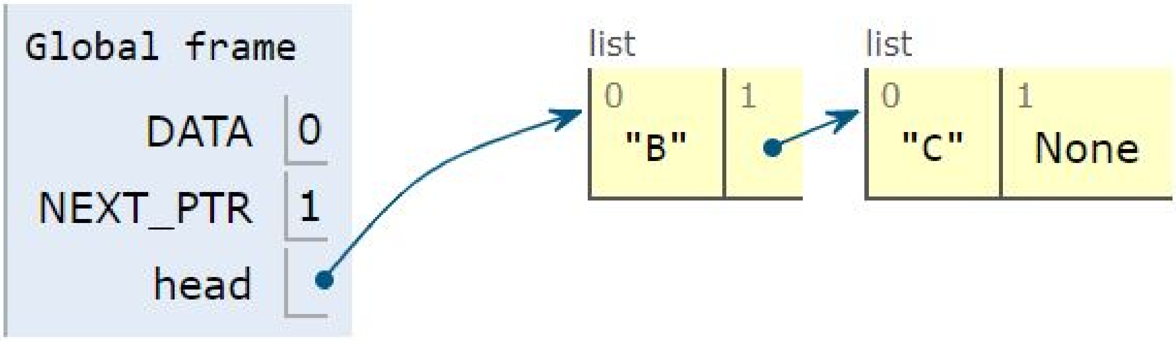
```
new_node[NEXT_PTR] = head
head = new_node
```



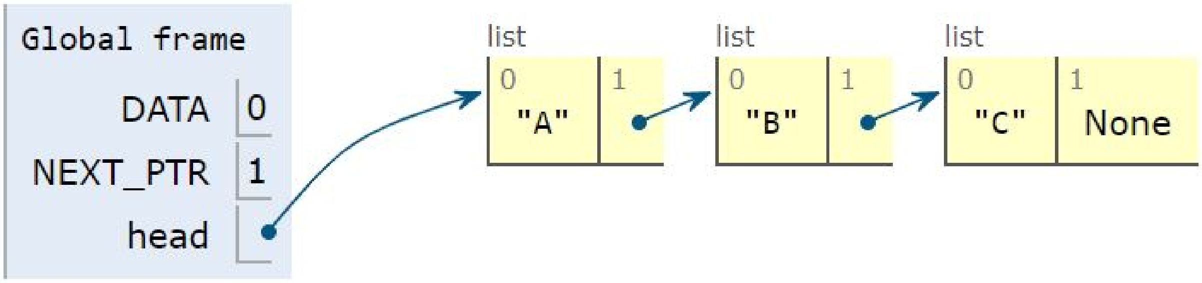
Remove at Head



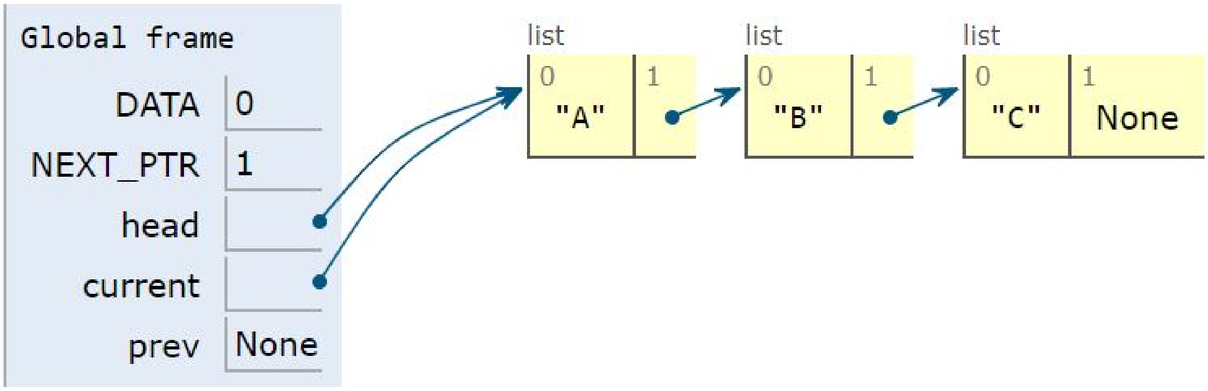
```
if head is not None:
    head = head[NEXT_PTR]
```



Reverse Linked List



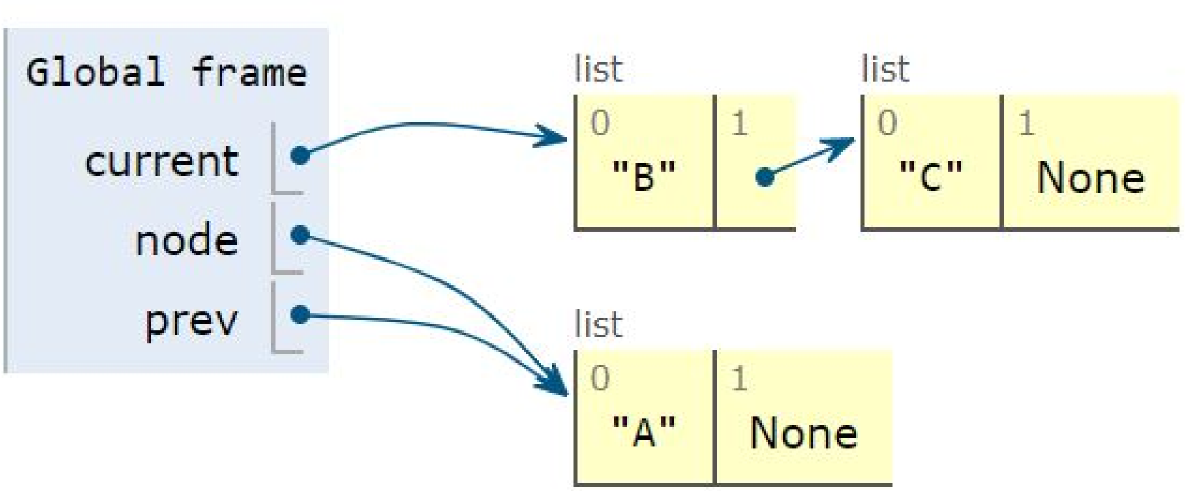
```
current = head
prev    = None
```



Reverse Linked List

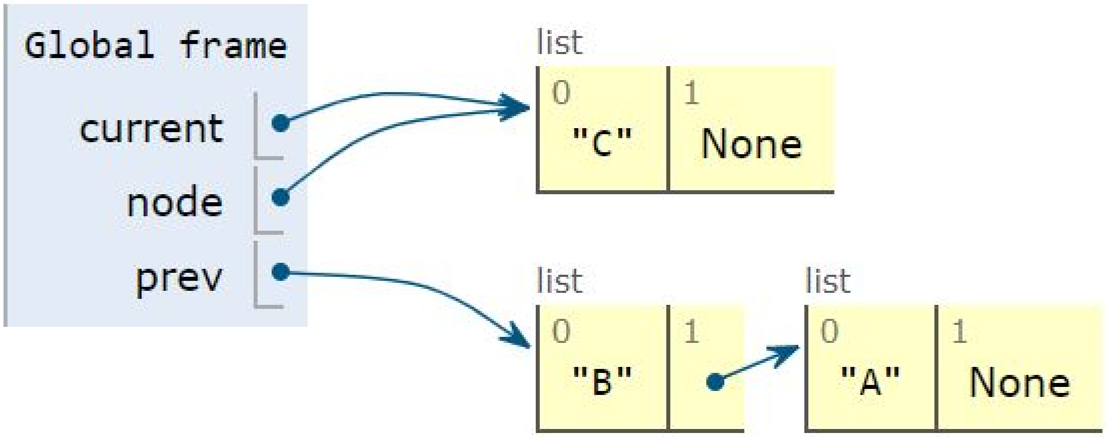
```
while current is not None:
    node              = current[NEXT_PTR]
    current[NEXT_PTR] = prev
    prev              = current
    current           = node
head = prev
```

- processing first node

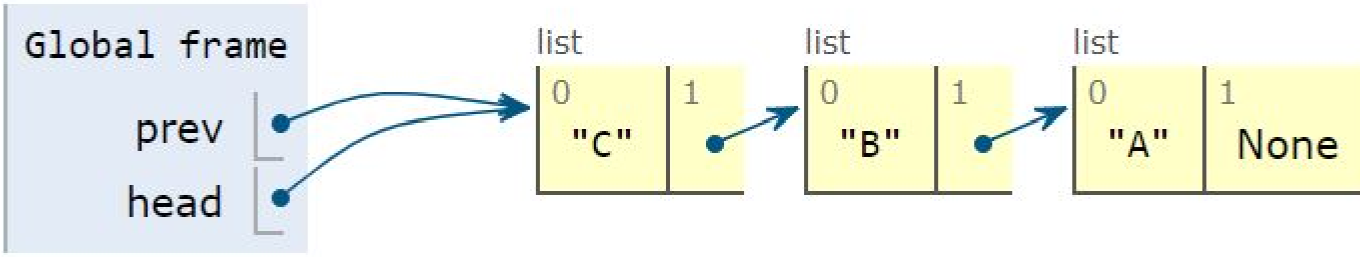


Reverse Linked List

- intermediate step



- final result

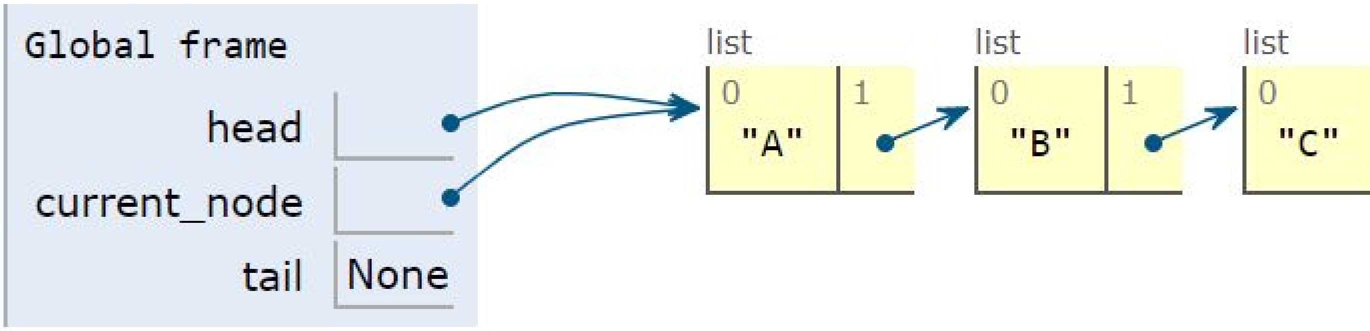


Reverse via Recursion

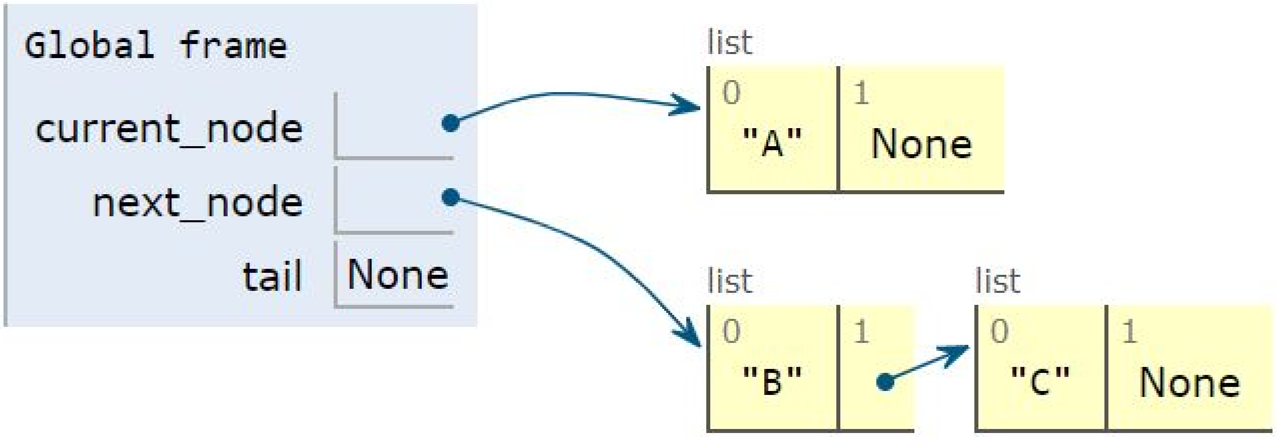
```
def rev_list(current_node, tail = None):
    next_node = current_node[NEXT_PTR]
    current_node[NEXT_PTR] = tail
    if next_node is None:
        return current_node
    else:
        return rev_list(next_node, current_node)

reverse_head = rev_list(head)
```

- split into a node and sub-list
- reverse sub-list (recursively)
- link sub-list and node in revese

Reverse via Recursion

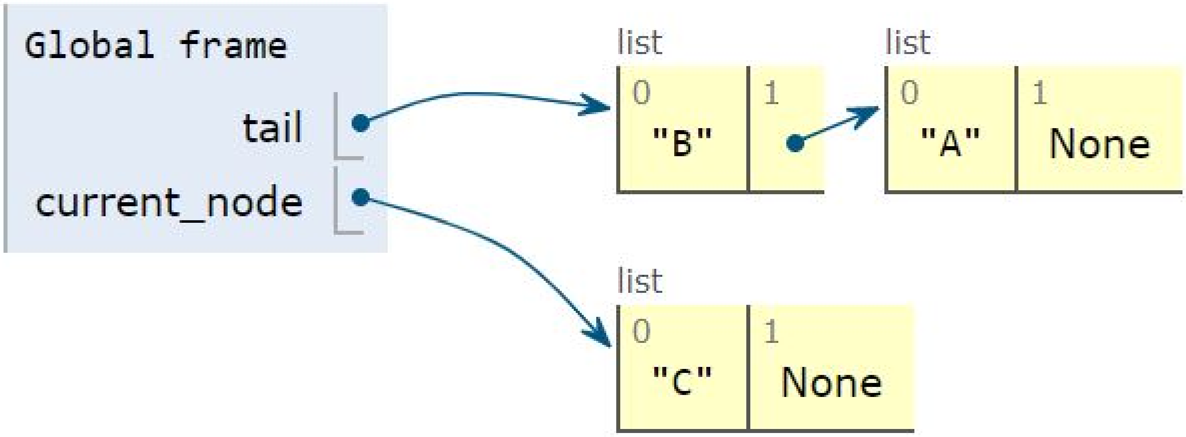


- split into node and sublist

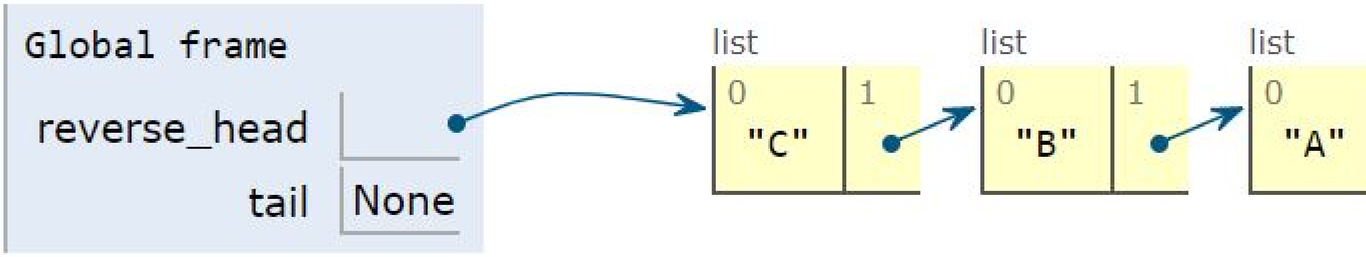


Reverse via Recursion

- reverse sublist (recursively)



- link parts in reverse



Code for Lists

```
DATA     = 0
NEXT_PTR = 1

def construct_list(data_list):
    head = None
    for e in data_list:
        new_node = [e, None]
        if head is None:
            head = new_node; tail = new_node
        else:
            tail[NEXT_PTR] = new_node;
            tail = new_node
    return head

def insert_at_head(head, data_item):
    new_node = [data_item, None]
    new_node[NEXT_PTR] = head
    head = new_node
    return head
```

Code for Lists (cont'd)

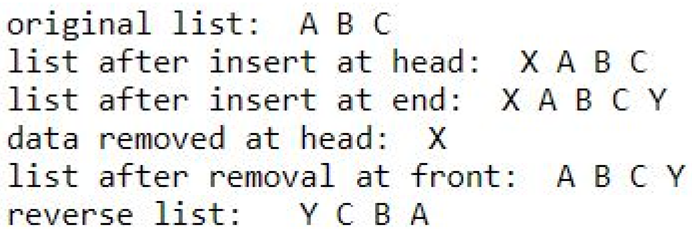
```
def insert_at_end(head, data_item):
    new_node = [data_item, None]
    if head is None:
        head = new_node
    else:
        tail = head
        while tail[NEXT_PTR] is not None:
            tail = tail[NEXT_PTR]
        tail[NEXT_PTR] = new_node
    return head

def remove_at_head(head):
    if head is not None:
        data_item = head[DATA]
        head = head[NEXT_PTR]
    else:
        data_item = None
    return data_item, head
```

Code for Lists (cont’d)

```
def rev_list(current_node, tail = None):
    next_node = current_node[NEXT_PTR]
    current_node[NEXT_PTR] = tail
    if next_node is None:
        return current_node
    else:
        return rev_list(next_node, current_node)

def traverse_list(first_node):
    next_node = first_node
    while next_node is not None:
        print(next_node[DATA], end=’ ’)
        next_node = next_node[NEXT_PTR]
    return
```

Code for Lists (cont’d)

```
head = construct_list([’A’,’B’,’C’])
print(’ original list: ’, end=’ ’)
traverse_list(head)
head = insert_at_head(head, ’X’)
print(’\n list after insert at head: ’, end=’ ’)
traverse_list(head)
head = insert_at_end(head, ’Y’)
print(’\n list after insert at end: ’, end=’ ’)
traverse_list(head)
data, head = remove_at_head(head)
print(’\n data removed at head: ’, data)
print(’ list after removal at front: ’, end=’ ’)
traverse_list(head)
rev_head = rev_list(head)
print(’\n reverse list:  ’, end = ’ ’)
traverse_list(rev_head)
```



```
original list:  A B C
list after insert at head:  X A B C
list after insert at end:  X A B C Y
data removed at head:  X
list after removal at front:  A B C Y
reverse list:   Y C B A
```