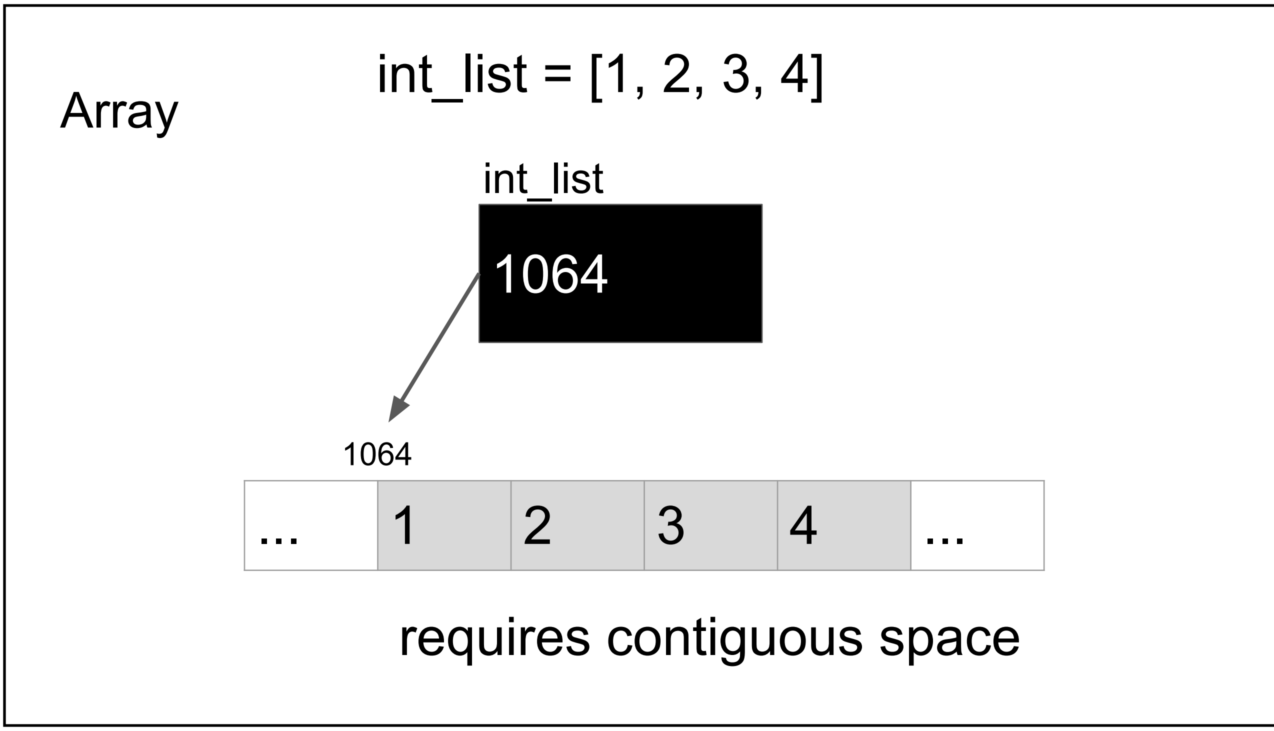
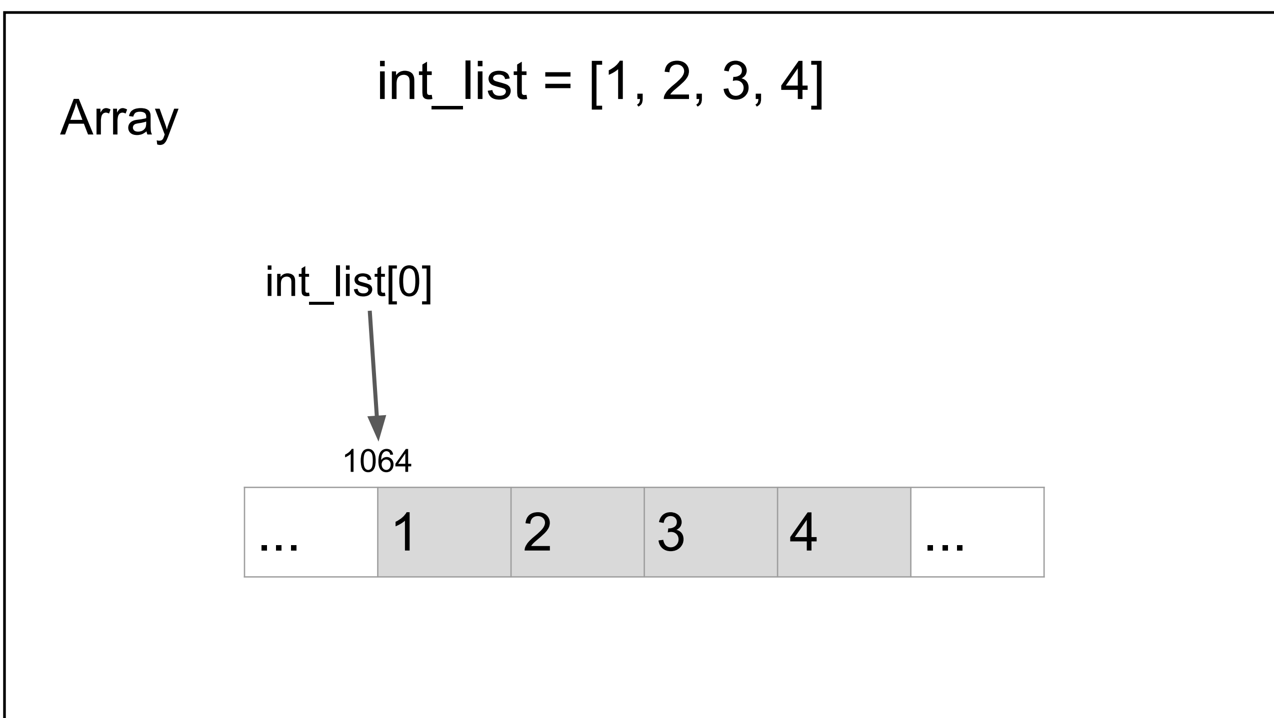
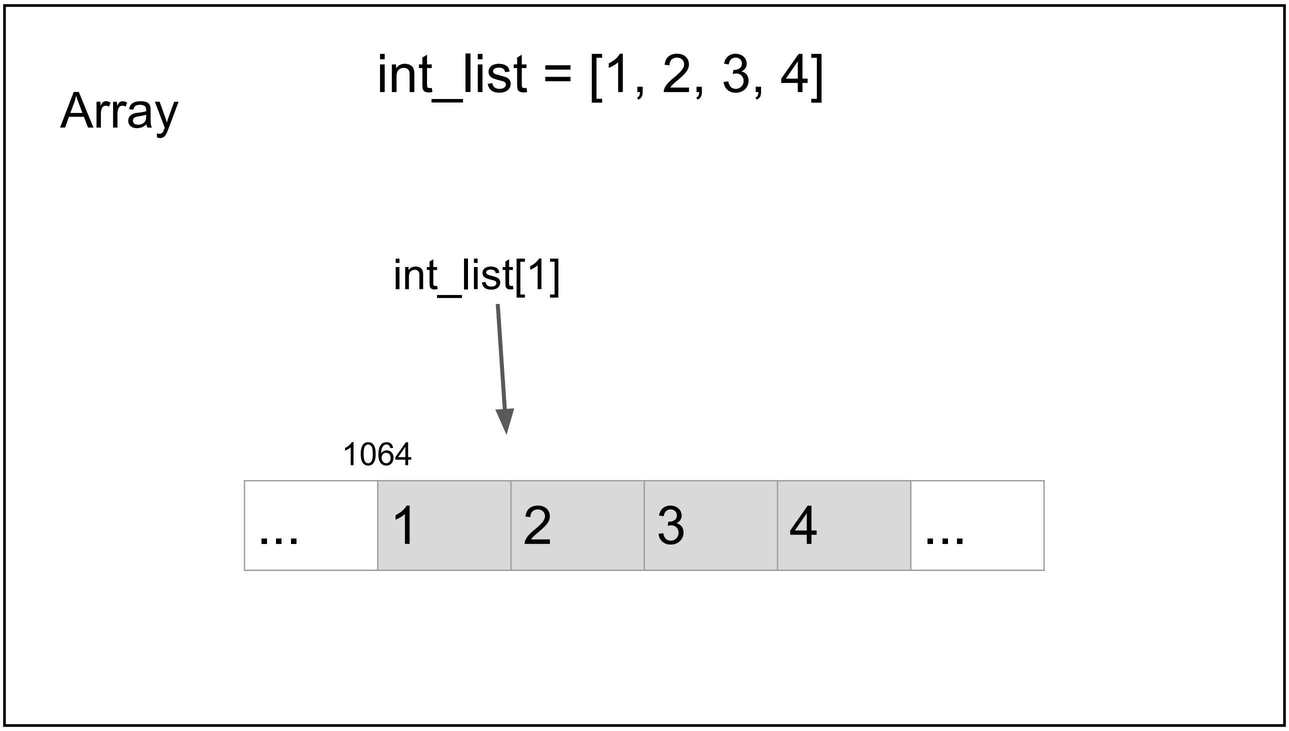
List

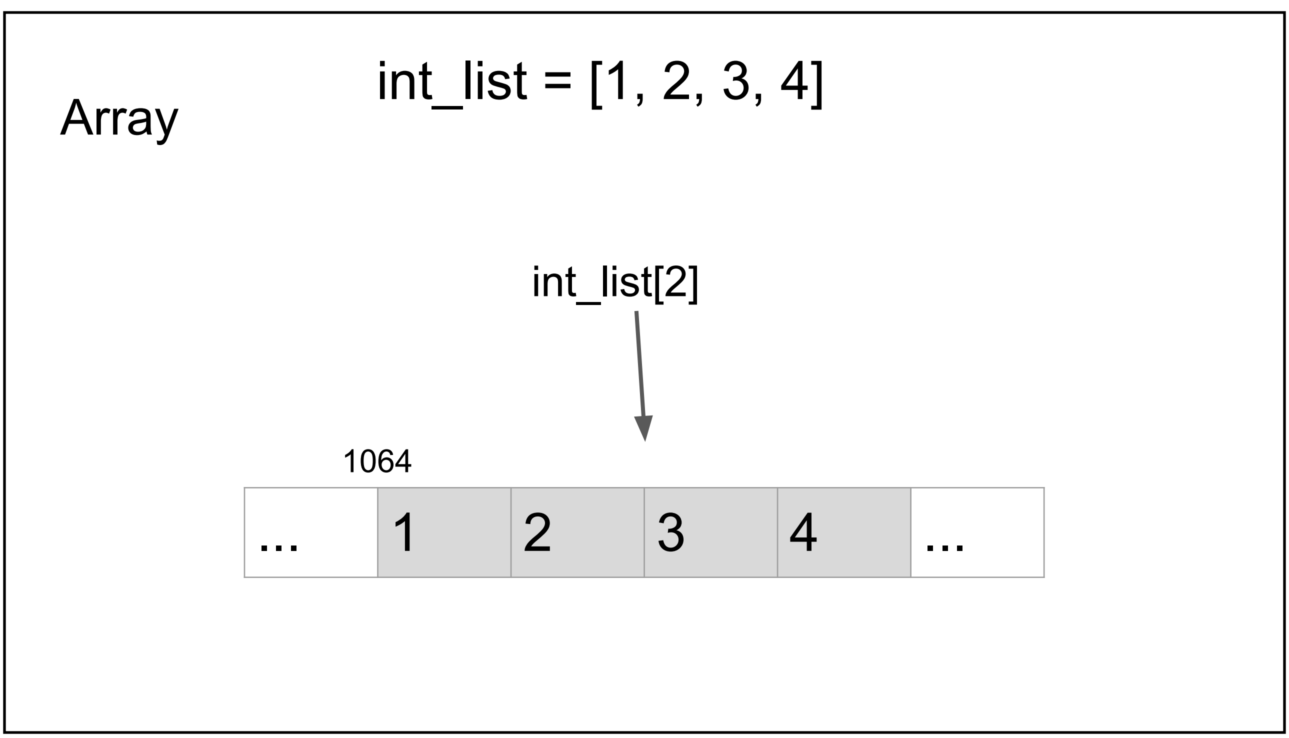
CPE202
Winter 2020

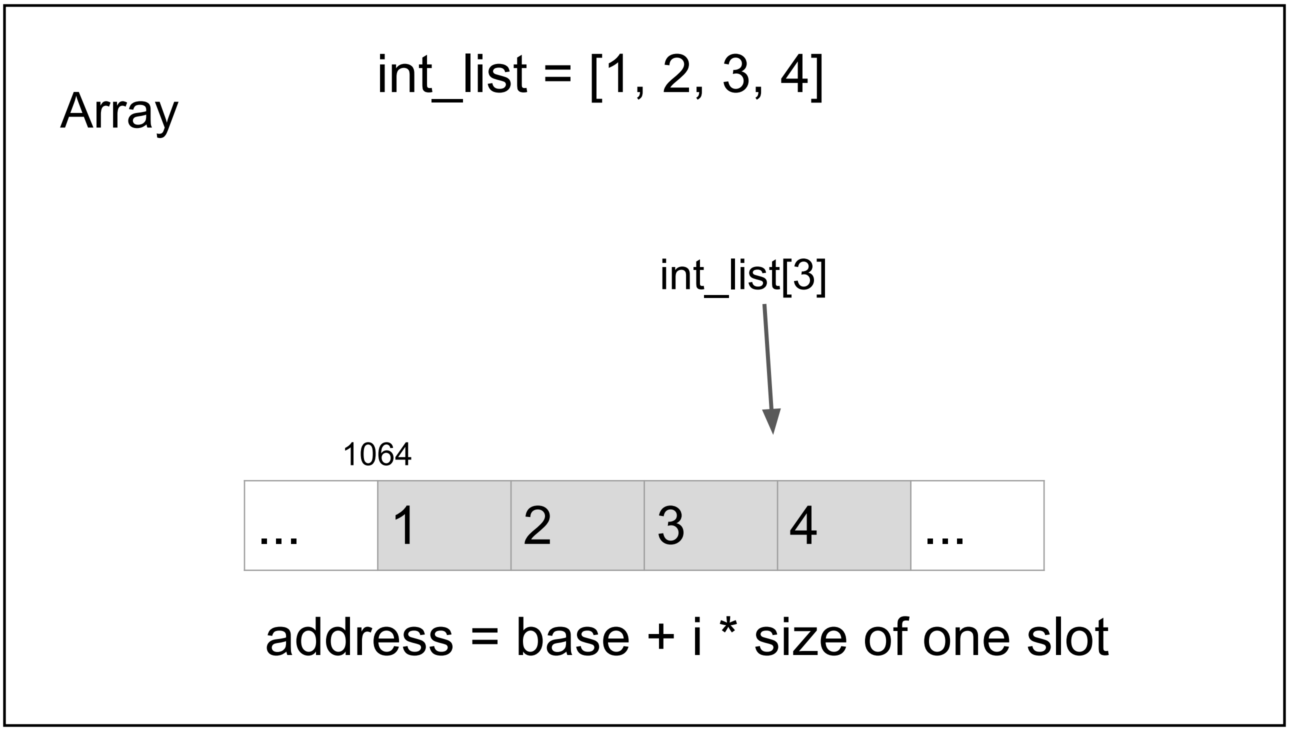
- Linear Data Structure
- collection of data

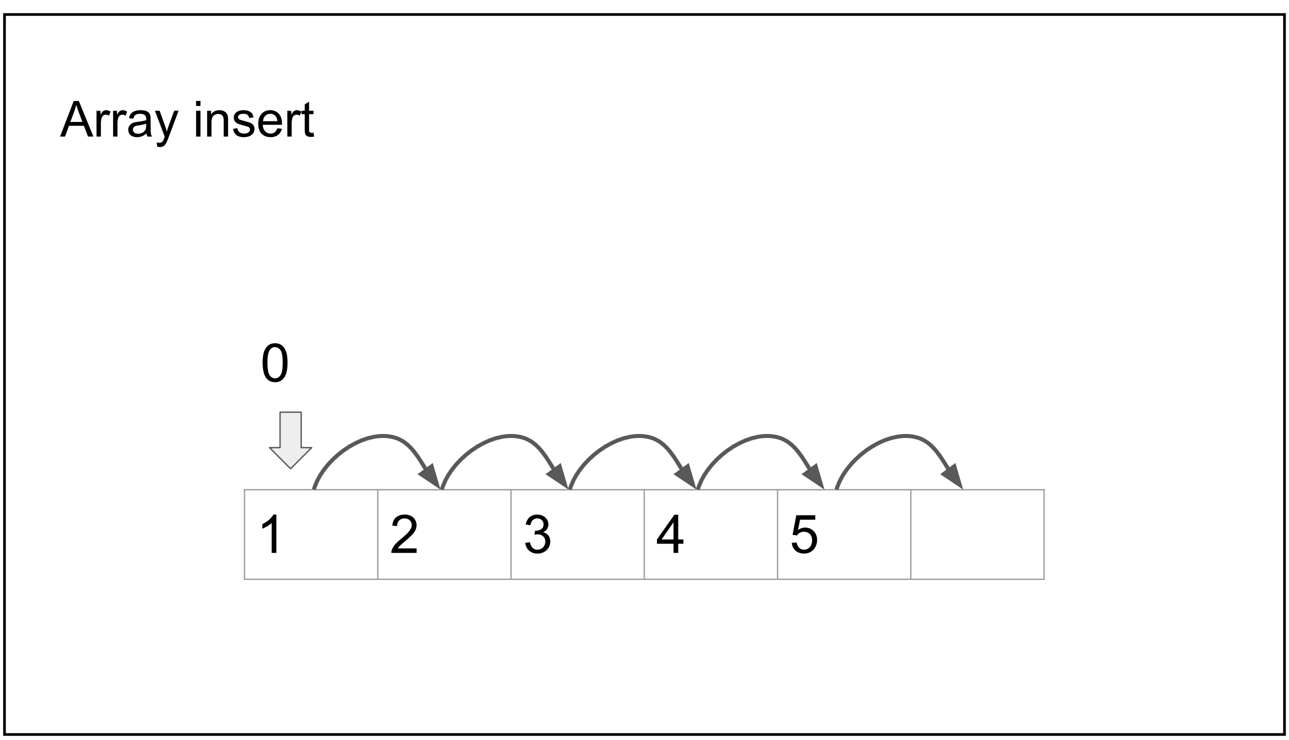

int_list = [1, 2, 3, 4]
Array
int_list
1064
1064
...
1
2
3
4
...
requires contiguous space


int_list = [1, 2, 3, 4]
Array
int_list[0]
1064
...
1
2
3
4
...


Array
int_list = [1, 2, 3, 4]
int_list[1]
1064
...
1
2
3
4
...

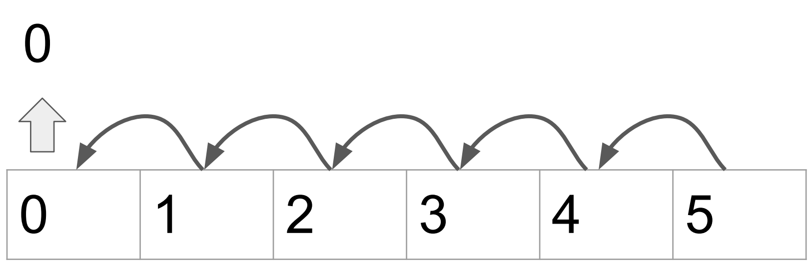

Array
int_list = [1, 2, 3, 4]
int_list[2]
1064
...
1
2
3
4
...


Array
int_list = [1, 2, 3, 4]
int_list[3]
1064
... 1 2 3 4 ...
address = base + i * size of one slot

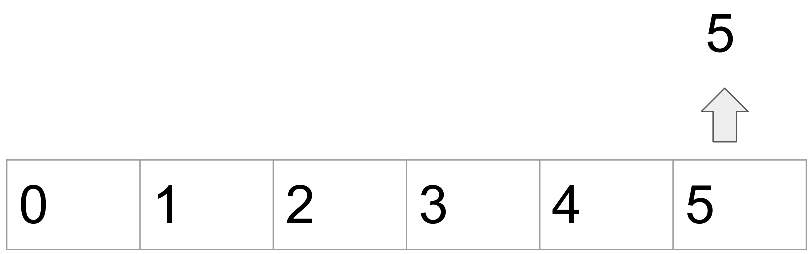

Array insert
0
1 2 3 4 5

Array append (insert at the end)

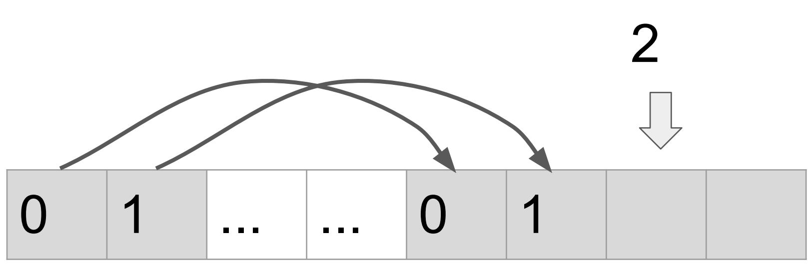
Array remove



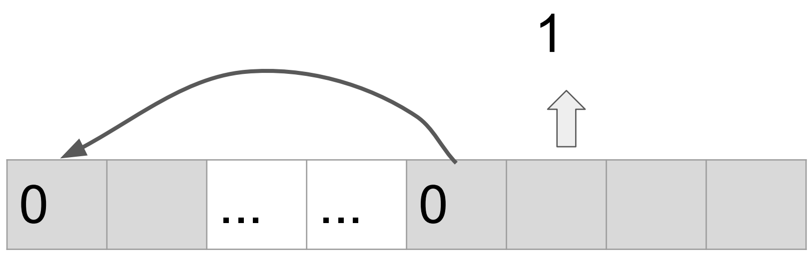
Array pop (remove from the end)



Array resize (enlarge)

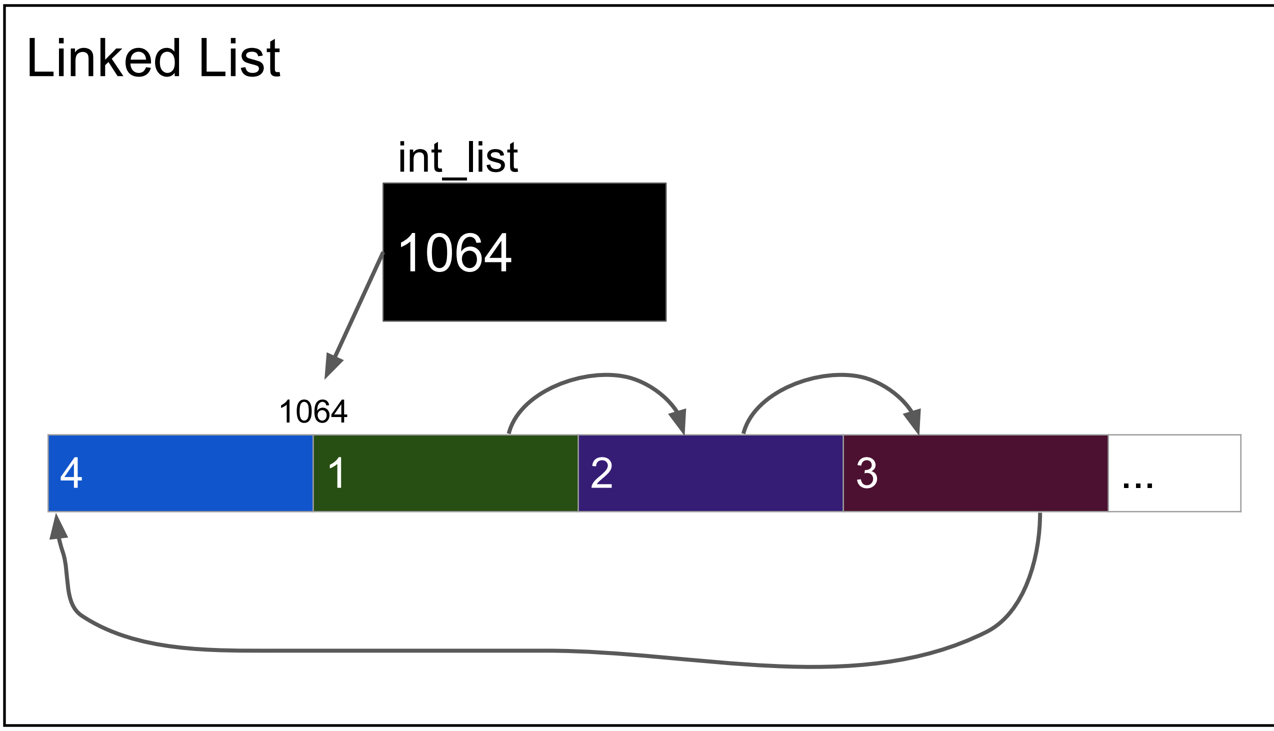


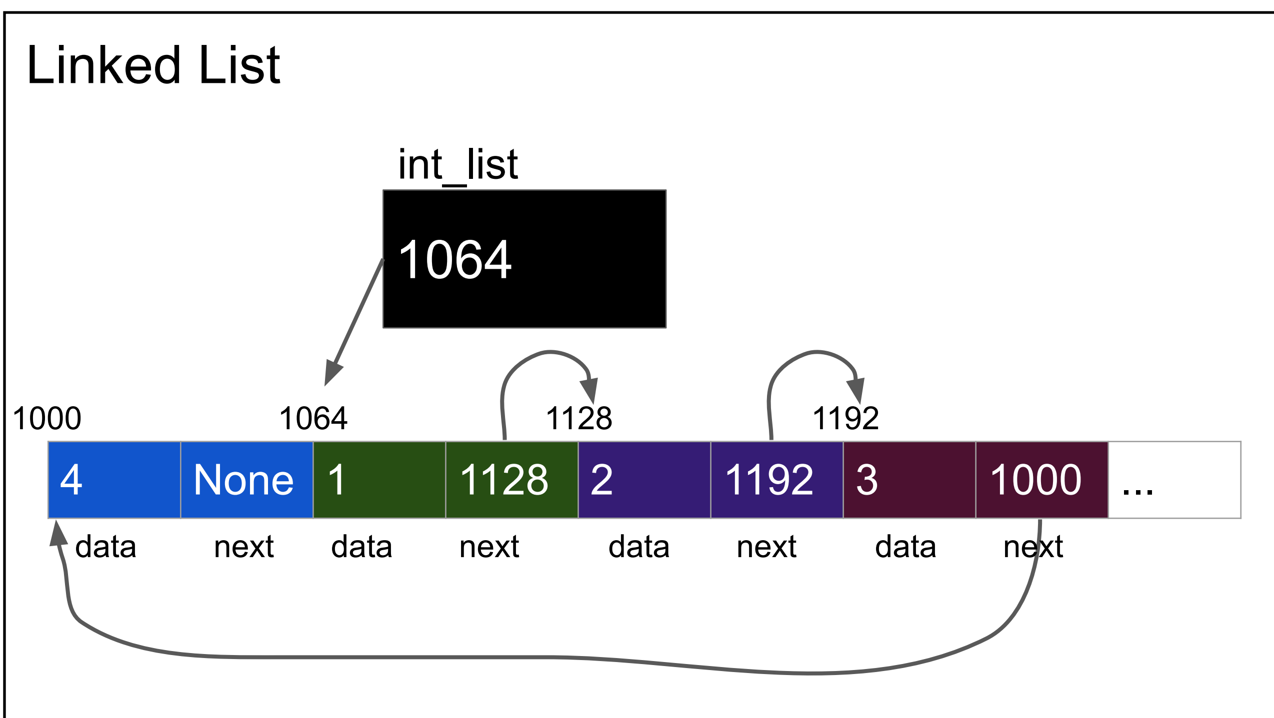
Array resize (shrink)

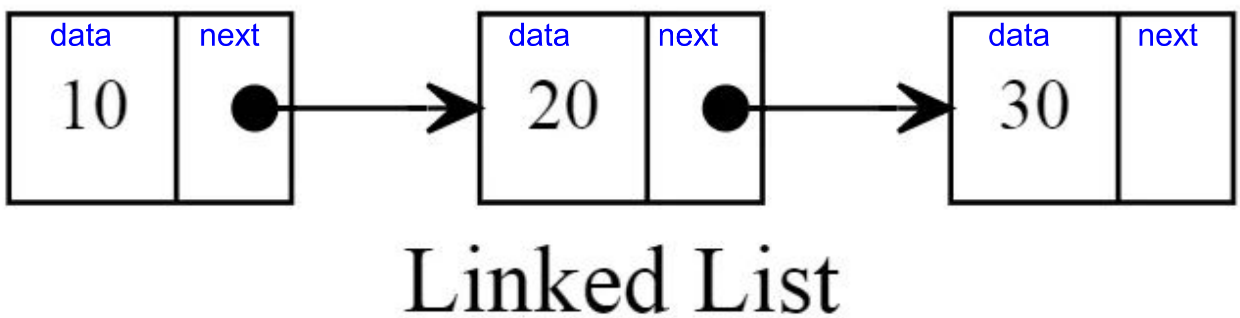


Time Complexity of Linked List v.s. Array

Operation	**Array**
Accessing i th element	O(1)
Appending an element	O(1)
Prepending an element	O(N)
Search	O(N) if not sorted
Deletion	O(N) Usually involves shifting items to the left
Resizing	O(N) on average in case of dynamic array

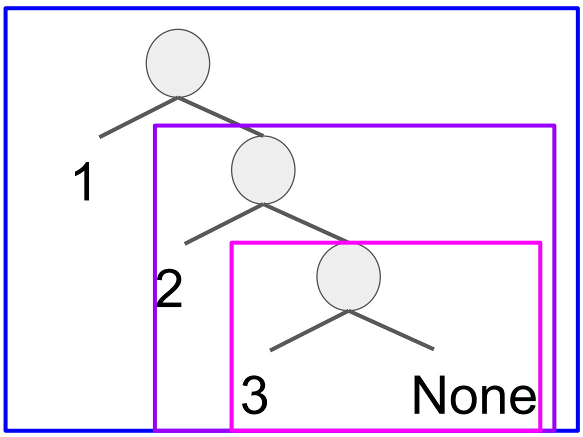

Linked List
int_list
1064
1064
4
1
2
3
...


Linked List
int_list
1064
1000
1064
1128
1192
4
None
1
1128
2
1192
3
1000
...
data
next
data
next
data
next
data
next




```
class Node:
    '''An Int List is one of
        - None, or
        - Node(data, next) : An Int List
    Attributes:
        data (int) : int value in this example but can be any data type
        next (Node) : an object of Node class (an Int list)
    '''
    def __init__(self, data, nxt=None):
        self.data = data   # a number
        self.next = nxt   # a reference to the next node
```

```
int_list = Node(1, Node(2, Node(3, None)))
```



Printing an object on the screen

```
def __repr__(self):
	return "Node(%d, %s)" % (self.data, self.nxt)
```

checking equality between objects

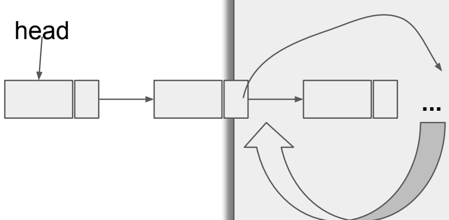
```
def __eq__(self, other):
        return isinstance(other, Node)\
            and self.data == other.data\
            and self.next == other.next
```

- Slower Access
    - only sequential access is possible

Operations on linked lists

```
def append_rec(int_list, data):
    if int_list is None:
        …
    else: #int_list is Node
        ...
```

```
def remove_rec(int_list, target)->Node:
    if int_list is None:
        ...
    elif int_list.data == target:
        ...
    else: #int_list.data != target
        ...
```



return None

```
def get_rec(int_list, pos)->int:
    if int_list is None:
        raise IndexError()
    if … :
        return int_list.data
    else:
        ...
```

count down

count up

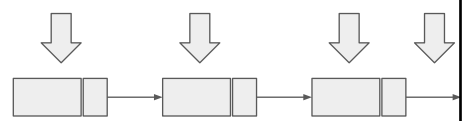
```
def find_rec(int_list, target)->int:
    return find_helper(int_list, target, ?)

def find_helper(int_list, target, pos):
    if int_list is None:
        ...
    elif ...:
        ...
    else:
        ...
```

```
def truncate(int_list, val)->Node:
    """truncates an IntList at the val.
    Args:
        int_list (Node): an IntList
        val (int): value to be found
    Returns:
        Node : an IntList (Node containing val) or None
    """
    if int_list == None:
        ...
    if val == int_list.data:
        ...
    ...
```

Iterator

```
def next(int_list)->(Node, int):
    """advances to the next element
    Args:
        int_list (Node): an IntList
    Returns:
        Node: the next element or None
        int: the current element's value
    Raises:
        StopIteration: when the end has been reached
    """
    if int_list is None:
        raise StopIteration
    return int_list.next, int_list.data
```



```
def sum_list(int_list):
    """Sums all the numbers in the list and returns the sum.

    Args:
        int_list (Node): None or An IntList
    Returns:
        int: the sum
    """
    if node == None:
        return ?
    return ?
```

```python
def sum_accum(int_list, acc=0):
    """Sums all the numbers in the list and returns the sum.

    Args:
        int_list (Node): None or IntList
        acc (int): accumulator
    Returns:
        int: the sum
    """
    if int_list is None:
        return ?
    return ?
```

Tail Recursion

- Tail Recursion
  - No computation after recursion.
  - Can be converted to iteration because the caller does not need to wait for the callee to return.
  - Some language compilers optimize tail recursion by converting it to iteration.
  - Python does not optimize it.

Tail Recursion

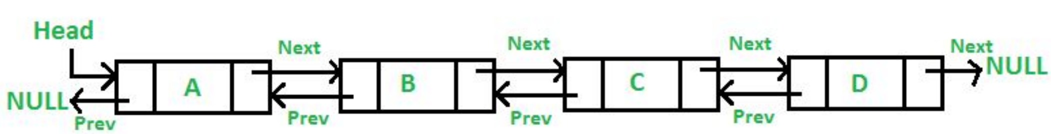
sum_list((1,(2,(3,(4,(5,None))))))	15
1 + sum_list((2,(3,(4,(5,None)))))	14
2 + sum_list((3,(4,(5,None))))	12
3 + sum_list((4,(5,None)))	9
4 + sum_list((5,None))	5
5 + sum_list(None)	0
sum_list(None)	

Non-Tail Recursive version

sum_accum((1,(2,(3,(4,(5,None))))), 0)
sum_accum((2,(3,(4,(5,None)))), 1)
sum_accum((3,(4,(5,None))), 3)
sum_accum((4,(5,None)), 6)
sum_accum((5,None), 10)
sum_accum(None, 15)
15

Tail Recursive version

Doubly Linked List



```
class Node:
    '''An Int List is one of
        - None, or
        - Node(data, next, prev) : An Int List
    Attributes:
        data (int) : int value in this example but can be any data type
        next (Node) : an object of Node class (an Int list)
        prev (Node) : an object of Node class (an Int list)
    '''
    def __init__(self, data, nxt=None, prev=None):
        self.data = data   # a number
        self.next = nxt   # a reference to the next node
        self.prev = prev   # a reference to the previous node
```

Time Complexity of Linked List v.s. Array

Operation	**Linked List**	**Array**
Accessing i th element	O(N)	O(1)
Appending an element	O(1) may require 2 variables	O(1)
Prepending an element	O(1) may require 2 variables	O(N)
Search	O(N) on average	O(N) if not sorted
Deletion	O(N) on average	O(N) Usually involves shifting items to the left
Resizing	-	O(N) on average in case of dynamic array