Lists
(bus)

Linked Lists
(train)
L.insert(0,-1)
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
</table>

L.insert(0,-1)
L.insert(0,-1)
L.insert(0,-1)

$O(n)$
L.append(9)
L.append(9)

O(1)?
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | X |

L.append(9)

O(1)?
find new memory and copy old contents

L.append(9)

O(n)
allocate twice as much memory as requested

L.append(9)

O(1), but sometimes O(n)
del L[0]
del L[0]
del L[0]

O(n)
del L[8]
del L[8]

Ω(1)
Lists

insert: \( O(n) \)

append: \( O(1), \) but sometimes \( O(n) \)

del: \( O(n) \)

access: ?
Lists

insert:    $O(n)$

append:   $O(1)$, but sometimes $O(n)$

del:       $O(n)$

access:    $O(1)$
Linked Lists

- Maine
- Idaho
- Utah

`data` `next` `head`
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

head = Node("Maine")
N1   = Node("Idaho")
N2   = Node("Utah")

# how do we connect Maine to Idaho to Utah?
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

head = Node("Maine")
N1   = Node("Idaho")
N2   = Node("Utah")

head.next = N1
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

head = Node("Maine")
N1   = Node("Idaho")
N2   = Node("Utah")

head.next = N1
N1.next   = N2
Linked Lists

```python
def print_list(node):
    while node:
        print(node.data)
        node = node.next
```

The diagram on the right illustrates a linked list with nodes labeled 'Maine', 'Idaho', and 'Utah', connected by arrows indicating the `next` pointers. The `head` node is located at the beginning of the list, and the algorithm `print_list` traverses the list by repeatedly printing the data of each node and moving to the next node until there are no more nodes to print.
node = head
while ???:
    print(node.data)
    ???
node = head
while ???:
    print(node.data)
    ???
node = head
while ???:
    print(node.data)
    node = node.next
node = head
while ???:
    print(node.data)
    node = node.next
node = head
while ???:
    print(node.data)
    node = node.next
node = head
while ???:
    print(node.data)
    node = node.next
node = head
while ???:
    print(node.data)
    node = node.next
Linked Lists

\[
\text{node} = \text{head} \\
\text{while} ???: \\
\quad \text{print}(\text{node.data}) \\
\quad \text{node} = \text{node.next}
\]
node = head

while node != None:
    print(node.data)
    node = node.next
Linked Lists: deleting (intuition)
Linked Lists: deleting (intuition)
Linked Lists: deleting (intuition)
Linked Lists: deleting (intuition)

O(1)
Linked Lists: access (intuition)
Linked Lists: access (intuition)

$O(n)$
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

    # print all elements in a list
    def printList(self):
        cur = self;
        while cur != None:
            print(cur.data, end=' ') # print space-delimited
            cur = cur.next
        print('') # print final carriage return

    # insert a node after the current node
    def insertAfter(self, node):

    # insert a node at the end of the list
    def insertAtEnd(self, node):
```python
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

    # insert a node after the current node
    def insertAfter(self, node):
        temp = self.next  # current next node
        self.next = node  # set current next to new node
        node.next = temp  # set new node's next to previous next

head = Node("A")
head.insertAfter( Node("B") )
head.insertAfter( Node("C") )
head.printList()
```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

    # insert a node after the current node
    def insertAfter(self, node):
        temp = self.next  # current next node
        self.next = node  # set current next to new node
        node.next = temp  # set new node's next to previous next

head = Node("A")
head.insertAfter(Node("B"))
head.insertAfter(Node("C"))
head.printList()
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

    # insert a node after the current node
    def insertAfter(self, node):
        temp = self.next  # current next node
        self.next = node  # set current next to new node
        node.next = temp  # set new node's next to previous next

head = Node("A")
head.insertAfter( Node("B") )
head.insertAfter( Node("C") )
head.printList()
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

    # insert a node after the current node
    def insertAfter(self, node):
        temp = self.next  # current next node
        self.next = node  # set current next to new node
        node.next = temp  # set new node's next to previous next

head = Node("A")
head.insertAfter( Node("B") )
head.insertAfter( Node("C") )
head.printList()
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

# insert a node after the current node
def insertAfter(self, node):
    temp = self.next # current next node
    self.next = node # set current next to new node
    node.next = temp # set new node's next to previous next

head = Node("A")
head.insertAfter( Node("B") )
head.insertAfter( Node("C") )
head.printList()
# --- DRILL ---
# write member functions insertAfter & insertAtEnd

class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

    # insert a node after the current node
    def insertAfter(self, node):
        temp = self.next  # current next node
        self.next = node  # set current next to new node
        node.next = temp  # set new node's next to previous next

head = Node("A")
head.insertAfter( Node("B") )
head.insertAfter( Node("C") )
head.printList()
# --- DRILL ---
# write member functions insertAfter & insertAtEnd

class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

    # insert a node after the current node
    def insertAfter(self, node):
        temp = self.next  # current next node
        self.next = node  # set current next to new node
        node.next = temp  # set new node's next to previous next

    # insert a node at the end of the list
    def insertAtEnd(self, node):
        cur = self;
        while( cur.next != None ):  # find last node
            cur = cur.next
            cur.insertAfter( node )  # insert node

head = Node("A")
head.insertAtEnd( Node("B") )
head.insertAtEnd( Node("C") )
head.insertAtEnd( Node("D") )
head.printList()
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

    # insert a node after the current node
    def insertAfter(self, node):
        temp = self.next # current next node
        self.next = node # set current next to new node
        node.next = temp # set new node's next to previous next

    # insert a node at the end of the list
    def insertAtEnd(self, node):
        cur = self;
        while( cur.next != None ): # find last node
            cur = cur.next
        cur.insertAfter( node ) # insert node

head = Node("A")
head.insertAtEnd( Node("B") )
head.insertAtEnd( Node("C") )
head.insertAtEnd( Node("D") )
head.printList()
Circular, doubly linked lists (with a sentinel)
Circular, doubly linked lists (with a sentinel)
Circular, doubly linked lists (with a sentinel)
Circular, doubly linked lists (with a sentinel)
Circular, doubly linked lists (with a sentinel)

class Node:
    def __init__(self, data):

class Sentinel_DLL:
    def __init__(self):
Circular, doubly linked lists (with a sentinel)

class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None

class Sentinel_DLL:
    def __init__(self):
        data
        / 
        /
Circular, doubly linked lists (with a sentinel)

class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None

class Sentinel_DLL:
    def __init__(self):
        self.sentinel = Node(None)
Circular, doubly linked lists (with a sentinel)

class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None

class Sentinel_DLL:
    def __init__(self):
        self.sentinel = Node(None)
        self.sentinel.next = self.sentinel
        self.sentinel.prev = self.sentinel
Circular, doubly linked lists (with a sentinel)

# Return a reference to the first node in the list, if there is one.
# If the list is empty, return None.

def first_node(self):

"""
# Return a reference to the first node in the list, if there is one.  
# If the list is empty, return None.

def first_node(self):
    if self.sentinel.next == self.sentinel:
        return None
    else:
        return self.sentinel.next
# Return a reference to the first node in the list, if there is one.  
# If the list is empty, return None.

def first_node(self):
    if self.sentinel.next == self.sentinel:
        return None
    else:
        return self.sentinel.next

D = Sentinel_DLL()
first = D.first_node()
Circular, doubly linked lists (with a sentinel)

insert “C” after “B”
Circular, doubly linked lists (with a sentinel)

[Diagram of a circular, doubly linked list with a sentinel]

insert “C” after “B”
Circular, doubly linked lists (with a sentinel)

I create a node C
Circular, doubly linked lists (with a sentinel)

1 create a node C
2 C.prev = B
Circular, doubly linked lists (with a sentinel)

1. create a node C
2. C.prev = B
3. C.next = D
Circular, doubly linked lists (with a sentinel)

1. create a node C
2. C.prev = B
3. C.next = D
4. B.next = C
Circular, doubly linked lists (with a sentinel)

1. create a node C
2. C.prev = B
3. C.next = D
4. B.next = C
5. D.prev = C
Circular, doubly linked lists (with a sentinel)

1. create a node C
2. C.prev = B
3. C.next = D
4. B.next = C
5. D.prev = C
def insert_after(self, x, data):
    y = Node(data)  # make a new Node object.
    z = x.next       # y goes between x and z
Circular, doubly linked lists (with a sentinel)

def insert_after(self, x, data):
    y = Node(data)  # make a new Node object.
    z = x.next      # y goes between x and z

    y.prev = x
    y.next = z
    x.next = y
    z.prev = y

[Diagram of a circular, doubly linked list with a sentinel]

sentinel
def insert_after(self, x, data):
    y = Node(data)    # make a new Node object.
    z = x.next        # y goes between x and z
    y.prev = x
    y.next = z
    x.next = y
    z.prev = y
Circular, doubly linked lists (with a sentinel)

def insert_after(self, x, data):
    y = Node(data)  # make a new Node object.
    z = x.next      # y goes between x and z
    y.prev = x
    y.next = z
    x.next = y
    z.prev = y
Circular, doubly linked lists (with a sentinel)

def insert_after(self, x, data):
    y = Node(data)  # make a new Node object.
    z = x.next      # y goes between x and z

    y.prev = x
    y.next = z
    x.next = y
    z.prev = y
Circular, doubly linked lists (with a sentinel)

def insert_after(self, x, data):
    y = Node(data)  # make a new Node object.
    z = x.next      # y goes between x and z
    y.prev = x
    y.next = z
    x.next = y
    z.prev = y
Circular, doubly linked lists (with a sentinel)

# Delete node x from the list.
def delete(self, x):

/   ...   ...   ...

sentinel

x
Circular, doubly linked lists (with a sentinel)

# Delete node x from the list.
def delete(self, x):

sentinel...
# Delete node x from the list.

def delete(self, x):
    x.prev.next = x.next
    x.next.prev = x.prev
Circular, doubly linked lists (with a sentinel)

# Delete node x from the list.
def delete(self, x):
    x.prev.next = x.next
    x.next.prev = x.prev
Circular, doubly linked lists (with a sentinel)

# Delete node x from the list.
def delete(self, x):
    x.prev.next = x.next
    x.next.prev = x.prev
Circular, doubly linked lists (with a sentinel)

# Delete node x from the list.

def delete(self, x):
    x.prev.next = x.next
    x.next.prev = x.prev
# Delete node x from the list.
def delete(self, x):
    x.prev.next = x.next
    x.next.prev = x.prev
# Delete node x from the list.

def delete(self, x):
   x.prev.next = x.next
   x.next.prev = x.prev
Circular, doubly linked lists (with a sentinel)

# Delete node \( x \) from the list.

def delete(self, x):
    x.prev.next = x.next
    x.next.prev = x.prev
Circular, doubly linked lists (with a sentinel)

# Insert a new node at the end of the list.
def append(self, data):
# Insert a new node at the end of the list.
def append(self, data):
    last_node = self.sentinel.prev
    self.insert_after(last_node, data)
Circular, doubly linked lists (with a sentinel)

# Find a node containing data, and return a reference to it.
def find(self, data):
    self.sentinel.data = data  # Store copy of data in the sentinel

    x = self.first_node()
    while x.data != data:
        x = x.next

    self.sentinel.data = None  # Restore the sentinel's data.

    if x == self.sentinel:
        return None  # data wasn't really in the list
    else:
        return x  # we found it in x, in the list
Circular, doubly linked lists (with a sentinel)

# Find a node containing data, and return a reference to it.

def find(self, data):

    self.sentinel.data = data  # Store copy of data in the sentinel

    x = self.first_node()
    while x.data != data:
        x = x.next

    self.sentinel.data = None  # Restore the sentinel's data.

    if x == self.sentinel:
        return None  # data wasn't really in the list
    else:
        return x  # we found it in x, in the list
Circular, doubly linked lists (with a sentinel)

# Find a node containing data, and return a reference to it.
def find(self, data):

    self.sentinel.data = data  # Store copy of data in the sentinel

    x = self.first_node()
    while x.data != data:
        x = x.next

    self.sentinel.data = None  # Restore the sentinel's data.

    if x == self.sentinel:
        return None  # data wasn't really in the list
    else:
        return x  # we found it in x, in the list
Circular, doubly linked lists (with a sentinel)

# Find a node containing data, and return a reference to it.
def find(self, data):
    self.sentinel.data = data  # Store copy of data in the sentinel

    x = self.first_node()
    while x.data != data:
        x = x.next

    self.sentinel.data = None  # Restore the sentinel's data.

    if x == self.sentinel:
        return None  # data wasn't really in the list
    else:
        return x  # we found it in x, in the list
Circular, doubly linked lists (with a sentinel)

# Find a node containing data, and return a reference to it.
def find(self, data):

    self.sentinel.data = data  # Store copy of data in the sentinel

    x = self.first_node()
    while x.data != data:
        x = x.next

    self.sentinel.data = None  # Restore the sentinel's data.

    if x == self.sentinel:
        return None  # data wasn't really in the list
    else:
        return x  # we found it in x, in the list
# Find a node containing data, and return a reference to it.
def find(self, data):

    self.sentinel.data = data  # Store copy of data in the sentinel

    x = self.first_node()
    while x.data != data:
        x = x.next

    self.sentinel.data = None  # Restore the sentinel's data.

    if x == self.sentinel:
        return None  # data wasn't really in the list
    else:
        return x  # we found it in x, in the list
Circular, doubly linked lists (with a sentinel)

# Return k\(^{th}\) node in the list

def random_access(self, k):

    counter = 0
    x = self.first_node()
    while counter < k and x != sentinel:
        x = x.next
        counter = counter + 1

    if x == self.sentinel:
        return None  # list has fewer than k elements
    else:
        return x  # return k\(^{th}\) node
Lists

insert: 0(n)
append: 0(1), but sometimes 0(n)
delete: 0(n)
find: 0(n)
access: 0(1)

Linked Lists

insert:
append:
delete:
find:
access:
Lists
insert: $O(n)$
append: $O(1)$, but sometimes $O(n)$
delete: $O(n)$
find: $O(n)$
access: $O(1)$

Linked Lists
insert: $O(1)$
append: 
delete: 
find: 
access:
Lists

insert:    $O(n)$
append:    $O(1)$, but sometimes $O(n)$
delete:    $O(n)$
find:      $O(n)$
access:    $O(1)$

Linked Lists

insert:    $O(1)$
append:    $O(1)$
delete:    
find:      
access:    
Lists
insert: \(O(n)\)
append: \(O(1)\), but sometimes \(O(n)\)
delete: \(O(n)\)
find: \(O(n)\)
access: \(O(1)\)

Linked Lists
insert: \(O(1)\)
append: \(O(1)\)
delete: \(O(1)\)
find: 
access:
Lists
    insert:  \(O(n)\)
    append:  \(O(1)\), but sometimes \(O(n)\)
    delete:  \(O(n)\)
    find:    \(O(n)\)
    access:  \(O(1)\)

Linked Lists
    insert:  \(O(1)\)
    append:  \(O(1)\)
    delete:  \(O(1)\)
    find:    \(O(n)\)
    access:  \(\)
Lists

- insert: $O(n)$
- append: $O(1)$, but sometimes $O(n)$
- delete: $O(n)$
- find: $O(n)$
- access: $O(1)$

Linked Lists

- insert: $O(1)$
- append: $O(1)$
- delete: $O(1)$
- find: $O(n)$
- access: $O(n)$