3 Feb 2020

Linked List

CPE111 – Programming with Data Structures

Reference

- Rance D. Necaise. Data Structures and algorithms using python. Chapter7. John Wiley&Sons,Inc., 2011
- Michael T.Goodrich, Roberto Tamassia, Michael H. Goodwasser. Data Structures and Algorithms in python. Chapter5. John Wiley&Sons,Inc. 2013

The **array** and **Python list** can be used to implement many different abstract data types (ADT). They both store data in linear sequence order and provide easy access to their elements (items).

However, an array requires a large chunk of memory capable of accommodating references to many elements. Also the size of an array is fixed and cannot change.

Each time an array is created, the program must allocate a block of memory large enough to store the entire array.

For large arrays, it can be difficult or impossible for the program to locate a block of memory into which the array can be stored.

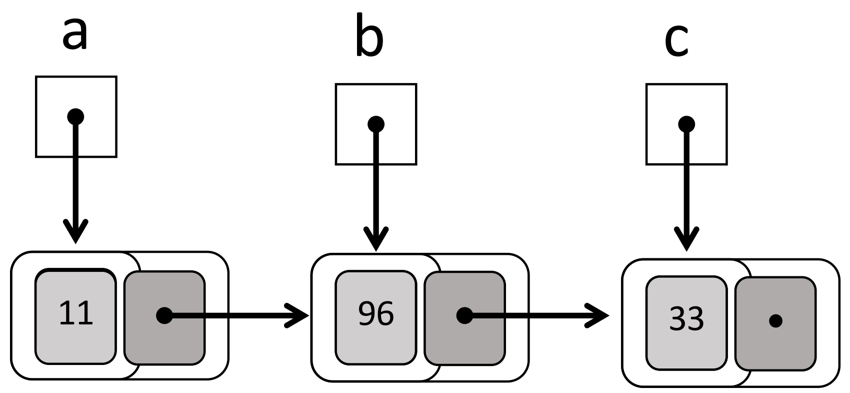
While the Python list does provide for an expandable space. That is highly optimized, and often a great choice for storage.

However, there are some notable disadvantages:

1. The length of a dynamic array might be longer than the actual number of elements that it stores.
2. Amortized bounds for operations may be unacceptable in real-time systems.
3. **Insertions** and **deletions** at interior positions of an array and Python list are expensive.

So, we introduce a data structure known as a **l i n k e d l i s t**, which is in contrast, relies on a more distributed representation in which a lightweight object, known as a node, is allocated for each element.

```
Class ListNode :
    def __init __(self, item):
        self._item = item
        self._next = None
```

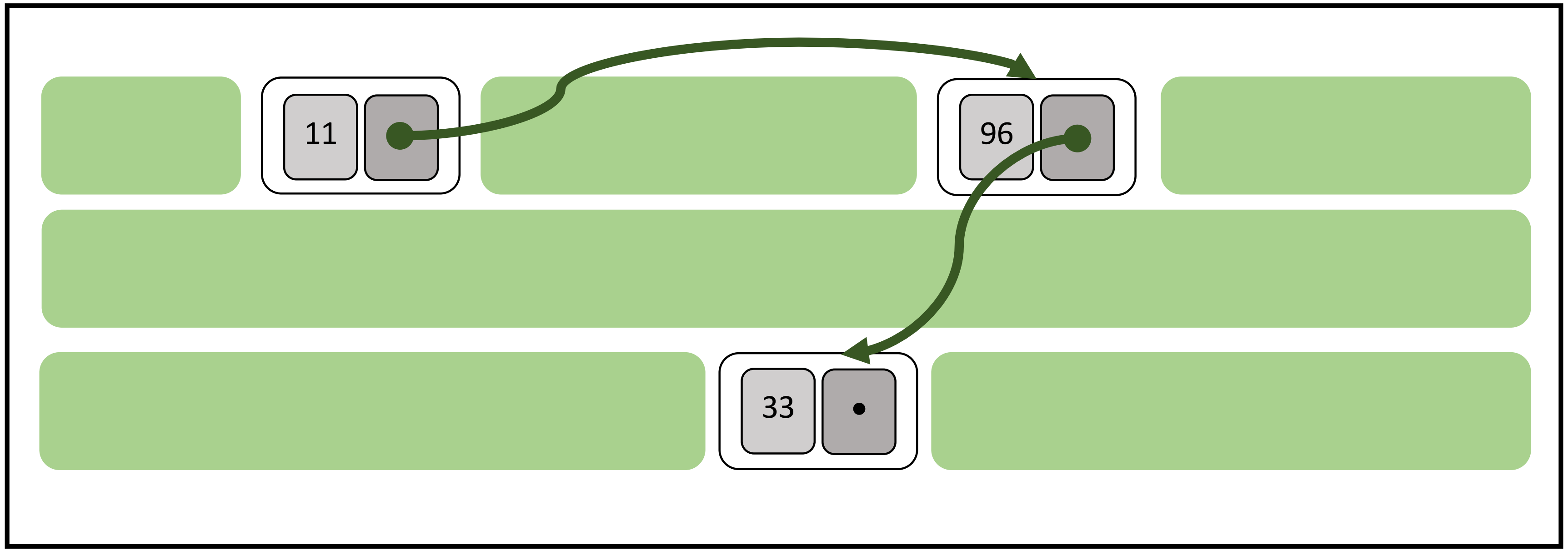


```
a = ListNode(11)
b = ListNode(96)
c = ListNode(33)
a._next = b
b._next = c
```

We can remove the external references **b** and **c**

```
print(a._item)              → 11
print(a._next._item)        → 96
print(a._next._next._item)  → 33
```

The Linked list stored in memory

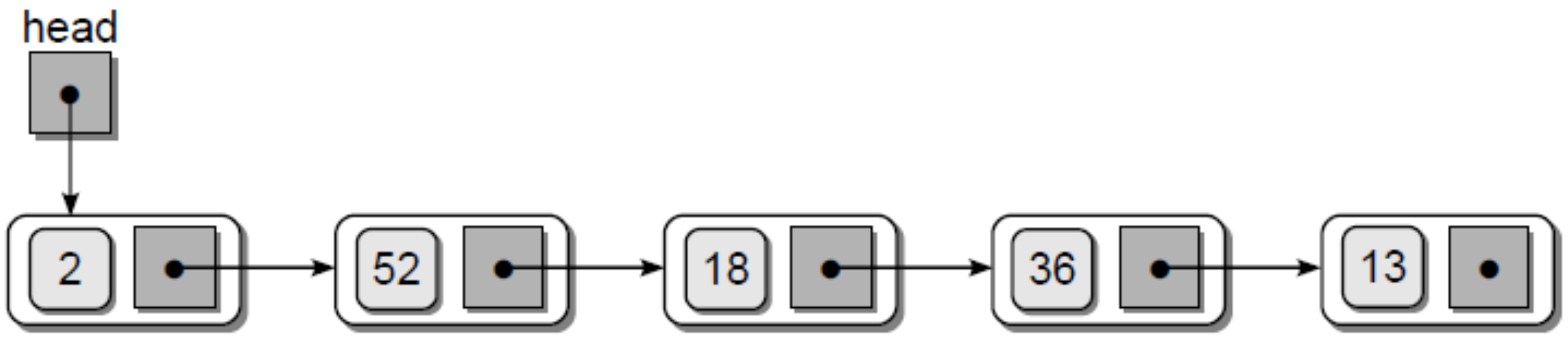


Linked list does not require to reserve a block of memory as array does

Singly Linked List

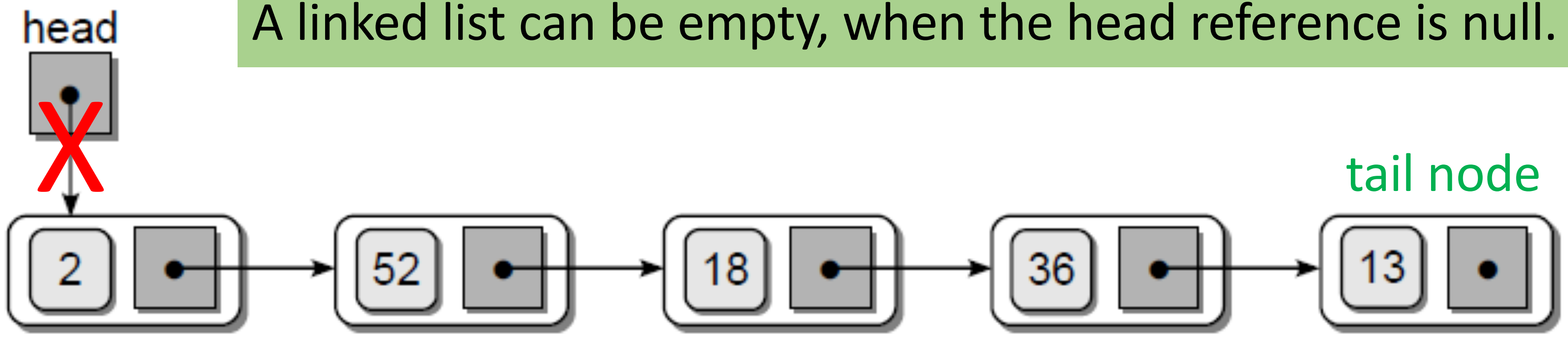
A singly linked list (SLlist) is a linked list in which each node contains a single link field and allows for a complete traversal from the first node to the last node.

The first node must be named or referenced by an external variable. This variable is commonly called “the head pointer”, or “head reference”.



The last node, commonly called the tail node, that is indicated by a null link (None) reference.

A linked list can be empty, when the head reference is null.

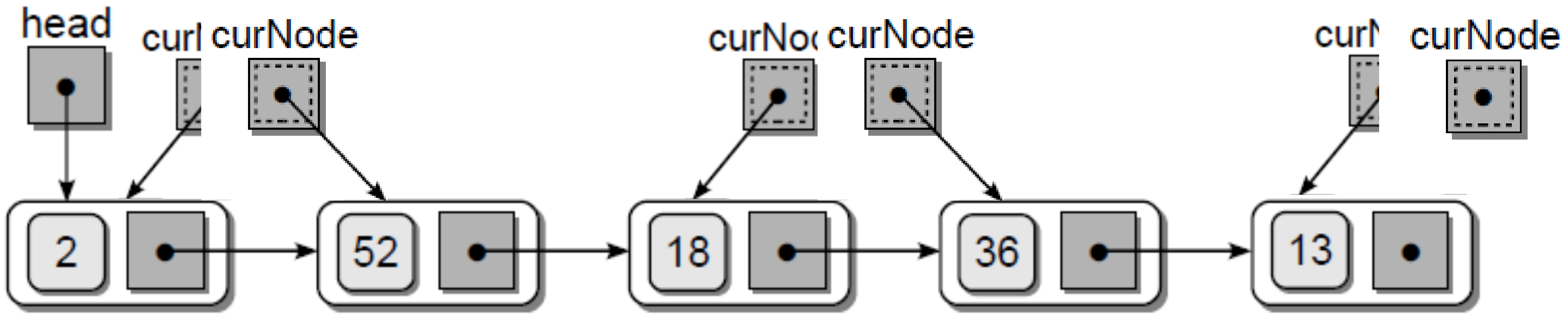


Most nodes in the list have no name and are simply referenced via the link field of the preceding node.

Traversing the node

```
2
52
18
36
13
```

```
# Traversing a linked list
def traverse(self):
    curNode = self._head
    while curNode is not None:
        print(curNode._item)
        curNode = curNode._next
```



Searching for a node

```
# Traversing a linked list
def traverse(self):
    curNode = self._head
    while curNode is not None:
        print(curNode._item)
        curNode = curNode._next
```

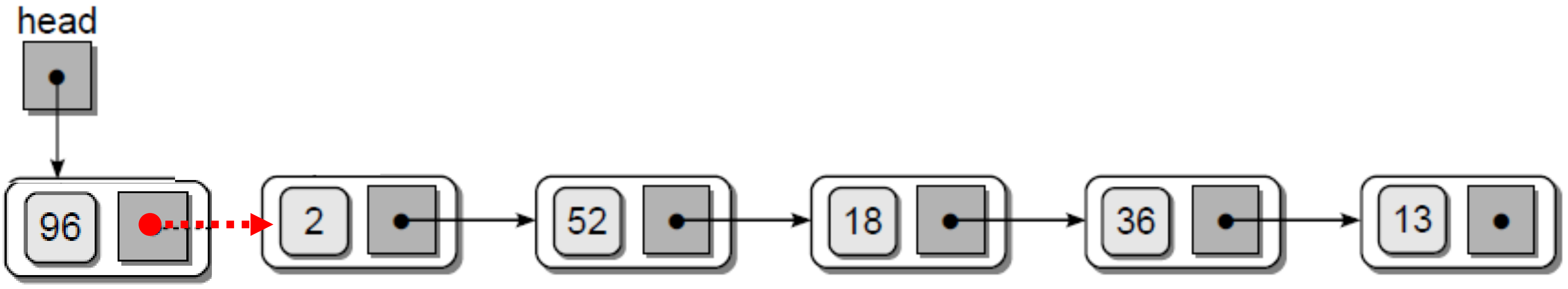
```
# Searching the target along a linked list
def search(self,target):
    curNode = self._head
    while curNode is not None and curNode._item != target:
        curNode = curNode._next
    return curNode is not None
```

The linked list search operation requires O(n) in the worst case.

Prepending a node

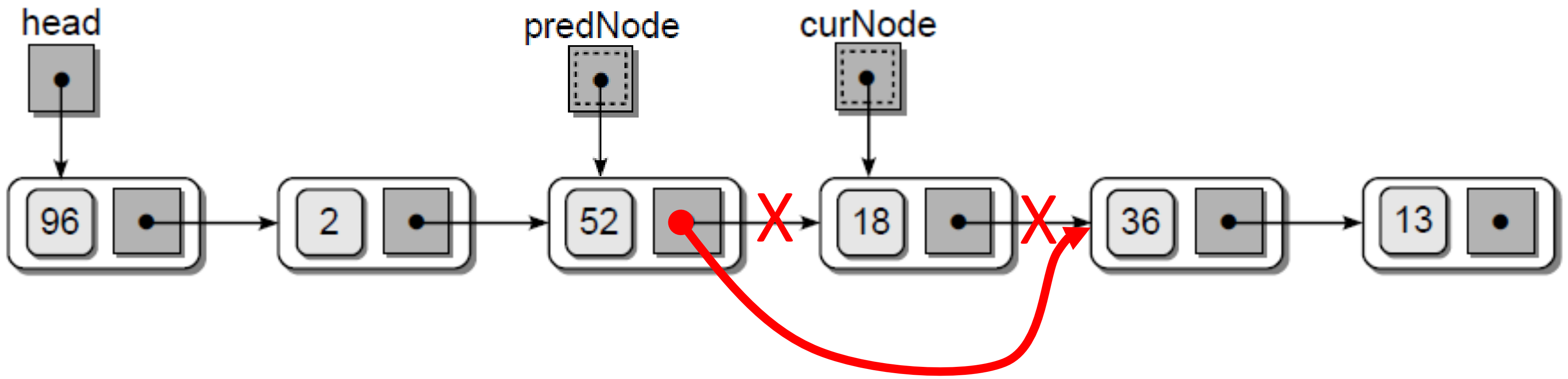
In case of an unordered list, we can simple prepend new items to the list. Prepending a node can be done in constant time since no traversal is required. since we can refer to the head reference.

```
# Prepending a node to the linked list
def prepend(self,item):
    newNode = ListNode(item)
    newNode._next = self._head
    self._head = newNode
```



Removing a node

```
# Remove a node from the linked list
def Remove(self,item):
    ...
    prevNode._next = curNode._next
    ...
```

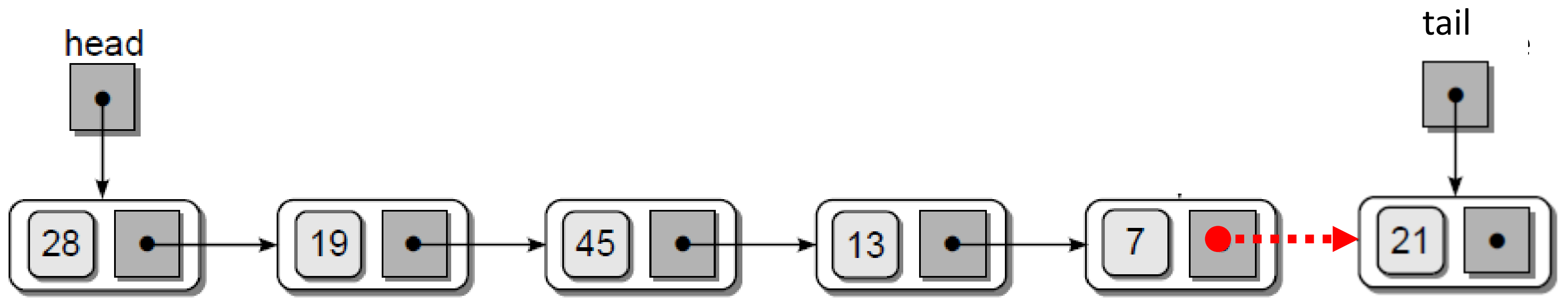


Appending a node

In some cases, we may need to append items to the end of the list instead. However the list has only a head reference so it requires linear time for traversal to reach the end of the list.

```
# Appending a node to the linked list
def append(self,item):
    newNode = ListNode(item)
    self._tail._next = newNode
    self._tail = newNode
```

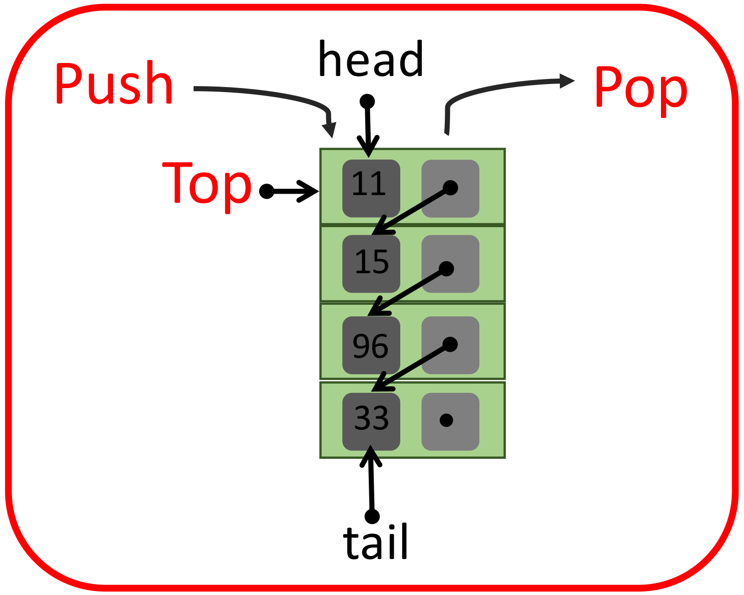
Instead of a single external head reference, we can use two external references, one for the head and one for the tail.

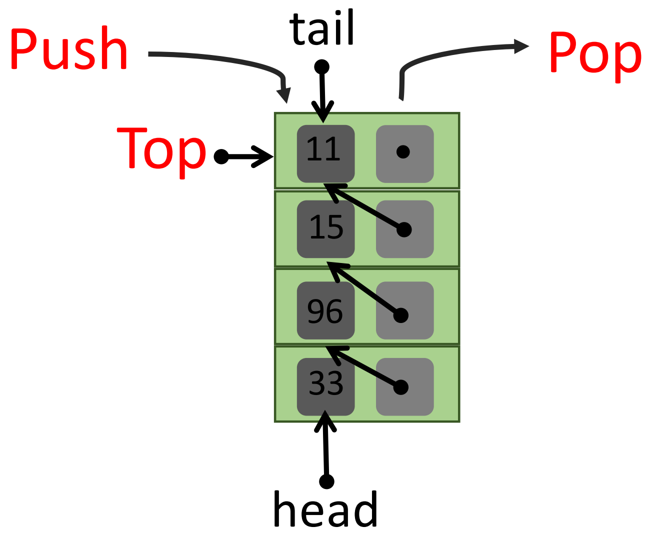


Implementing a Stack with a Singly Linked list

The stack needs only one top reference. we need to decide to model the top of the stack at the head or at the tail of the list.

Operation	Running Time
S.push(e)	$O(1)$
S.pop()	$O(1)$
S.top()	$O(1)$
len(S)	$O(1)$
S.is_empty()	$O(1)$

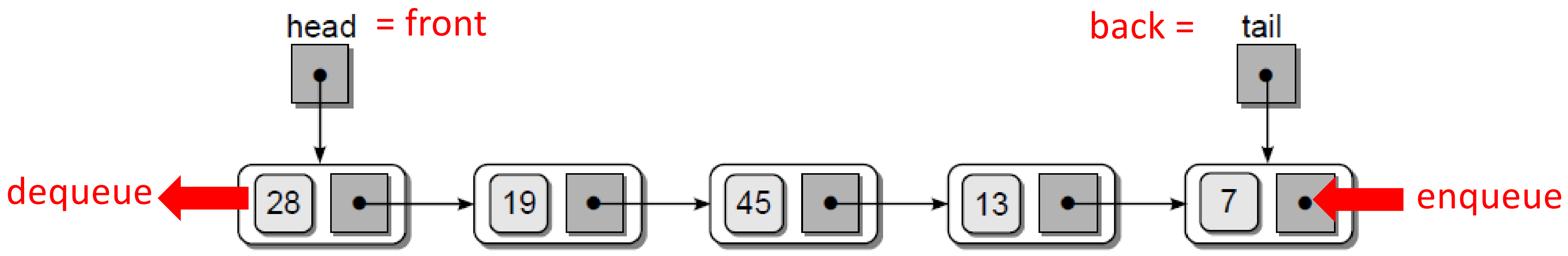




we can efficiently insert and delete items in constant time only at the head. Since all stack operations affect the top, we orient the top of the stack at the head of our list.

Implementing a Queue with a Singly Linked list

To implement the queue ADT while supporting worst-case $O(1)$-time for all operations. We need to perform operations on both ends of the queue, we need both a head reference and a tail reference .

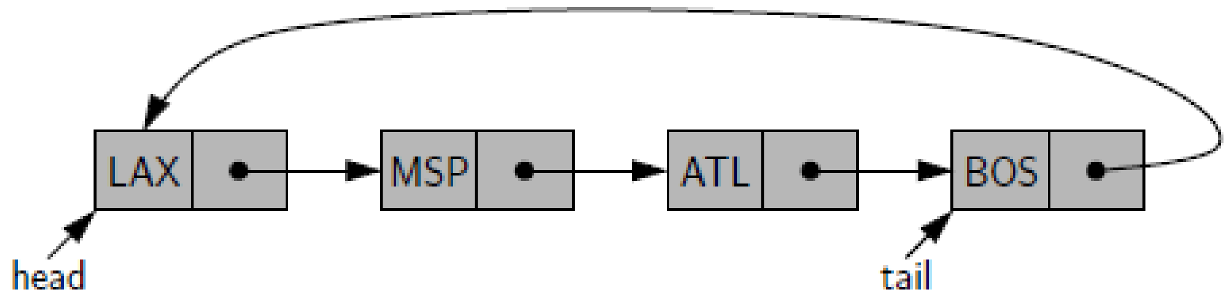


The natural orientation for a queue is to align the front of the queue with the head of the list, and the back of the queue with the tail of the list, because we must be able to enqueue elements at the back, and dequeue them from the front.

Circularly Linked List

In reality, the notion of a circular array was artificial, in that there was nothing about the representation of the array itself that was circular in structure. It was our use of modular arithmetic when "advancing" an index from the last slot to the first slot that provided such an abstraction.

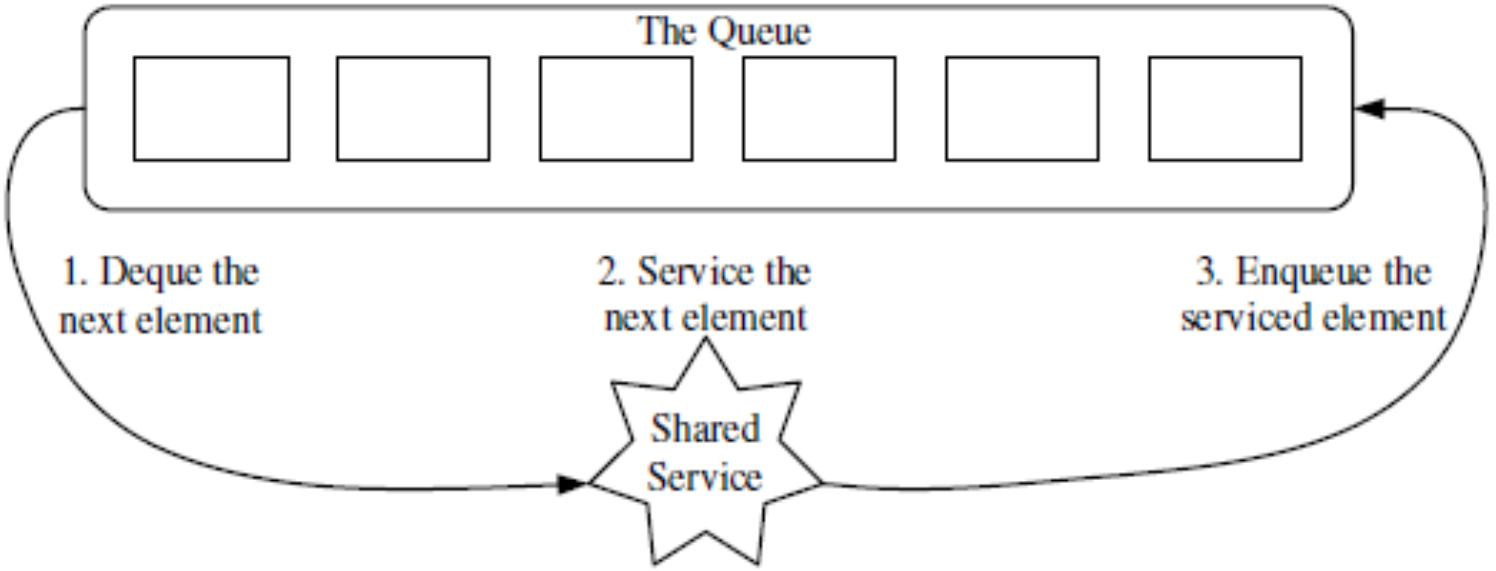
In the case of linked lists, there is a more tangible notion of a **circularly linked list**, as we can have the tail of the list use its next reference to point back to the head of the list



Round-Robin Schedulers

The application of a circularly linked list, we introduce a round-robin scheduler, which iterates through a collection of elements in a circular fashion and “services” each element by performing a given action on it.

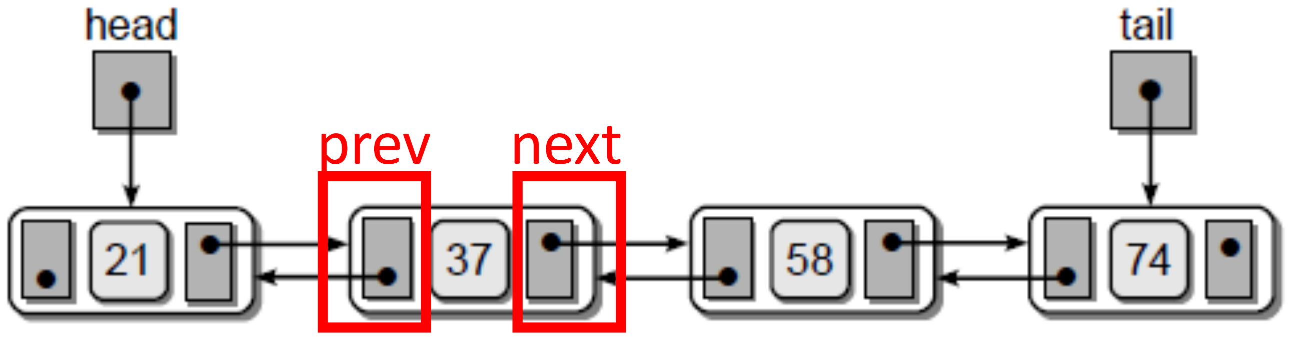
For instance, round-robin scheduling is often used to allocate slices of CPU time to various applications running concurrently on a computer.


The Queue
1. Deque the next element
2. Service the next element
3. Enqueue the serviced element
Shared Service

Doubly Linked List

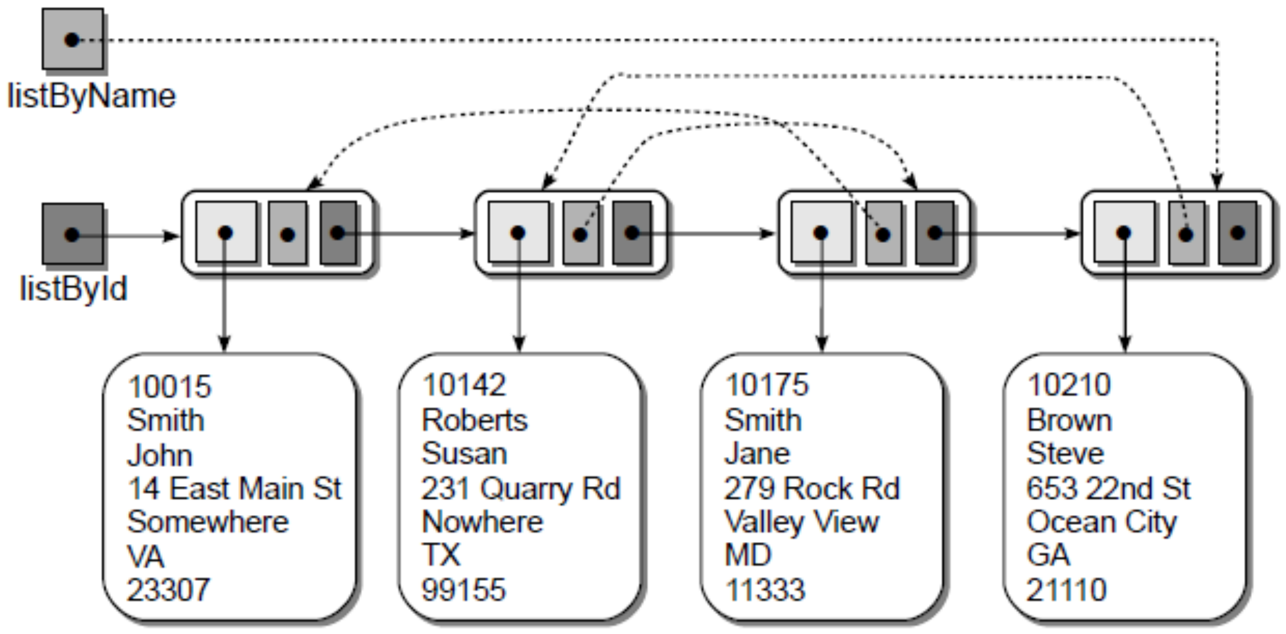
The singly linked list accesses and traversals begin with the first node and progress toward the last node but it cannot traverse the nodes in reverse order. Cause the problem, when you need to insert a new node immediately preceding that node. Since the predecessor of a node cannot be directly accessed.
For these operations, we need direct access to both the node following and preceding. Then we use **Doubly Linked List (DLlist)**

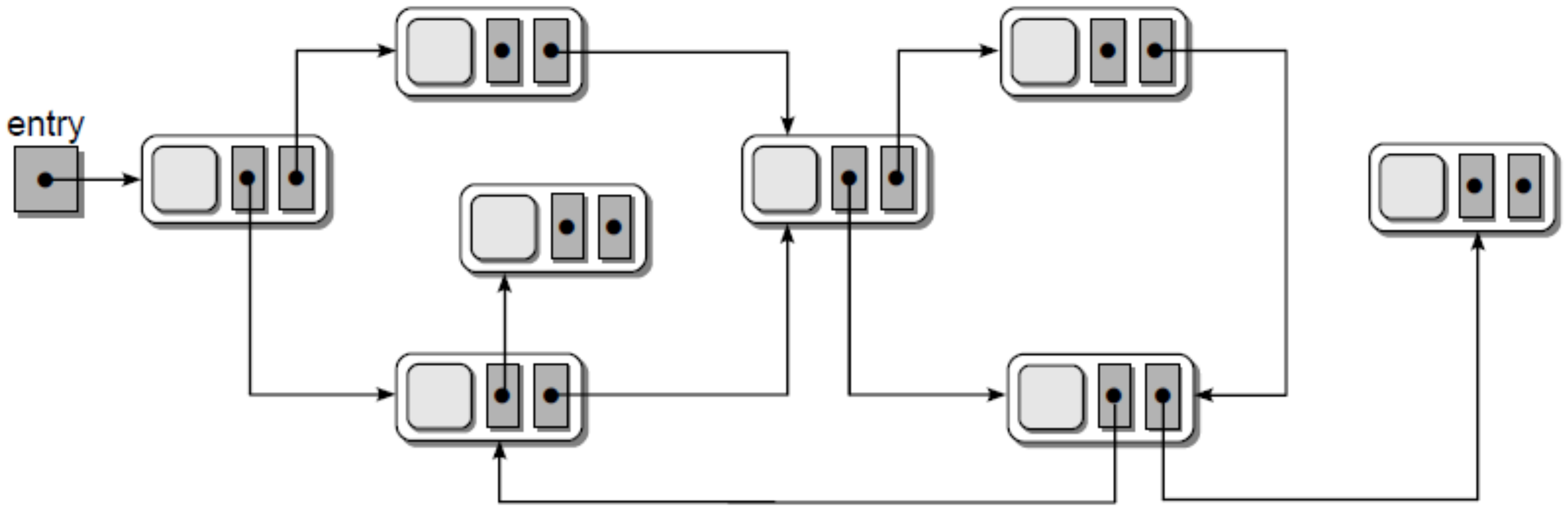
```
# Defines a Simple DlinkNode.
class DlinkNode(object):
    def __init__(self,item,prev,next):
        self._item = item
        self._prev = prev
        self._next = next
```

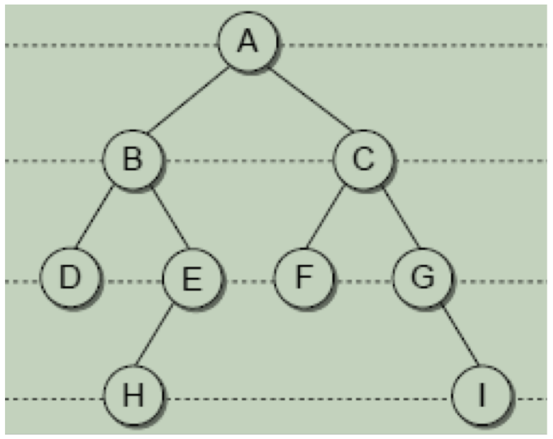


Multi-Linked List

In a multi-linked list, Multiple chains can be created using multiple keys or different data components to create the multiple links.



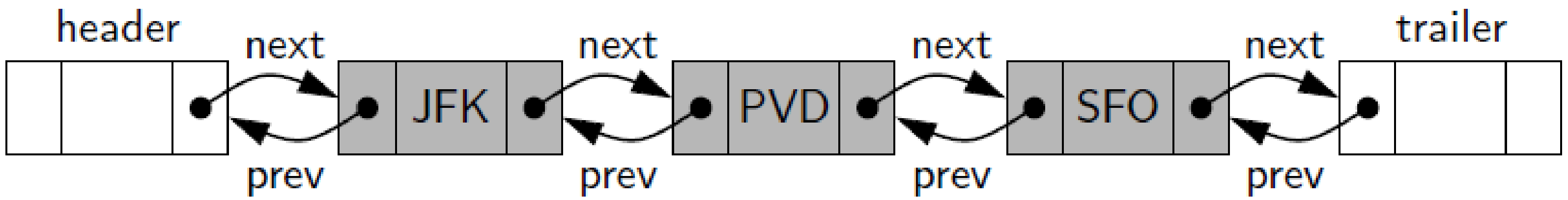




if more links are added to each node, we can connect the nodes to form any type of data structures e.g. The tree structure

Header and Trailer Sentinels

In order to avoid some special cases when operating near the boundaries of a doubly linked list, it helps to add special nodes at both ends of the list: a ***header*** node at the beginning of the list, and a ***trailer*** node at the end of the list. These “dummy” nodes are known as ***sentinels*** (or guards), and they do not store elements of the sequence.

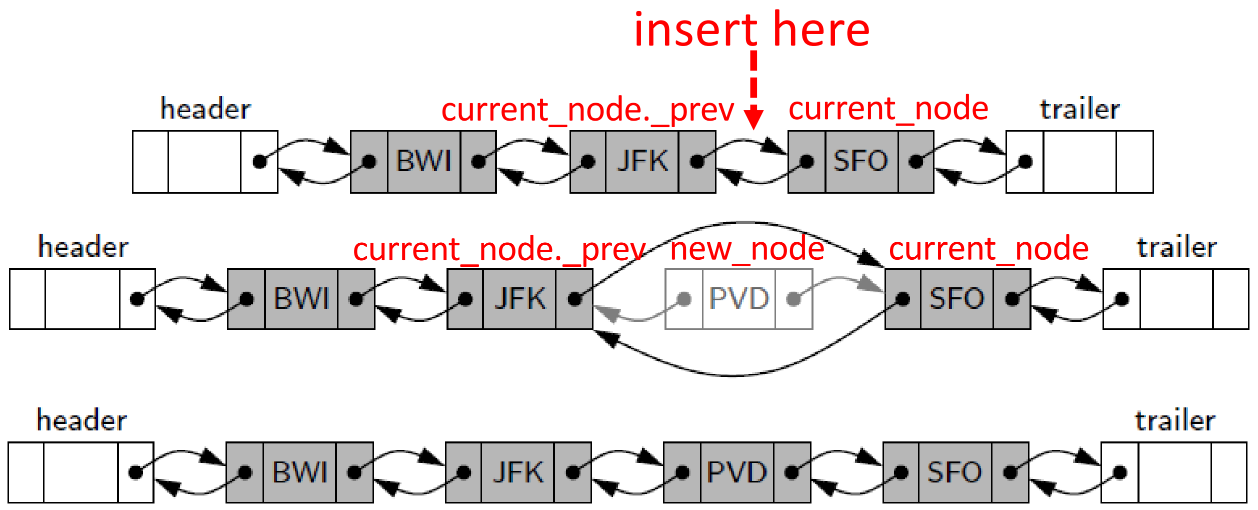


```
def __init__(self):
    self._header = DlinkNode(None,None,None)
    self._trailer = DlinkNode(None,None,None)
    self._header._next = self._trailer
    self._trailer._prev = self._header
```

advantage: we can treat all insertions in a unified manner, because a new node will always be placed between a pair of existing nodes.

Inserting a node from a doubly linked list

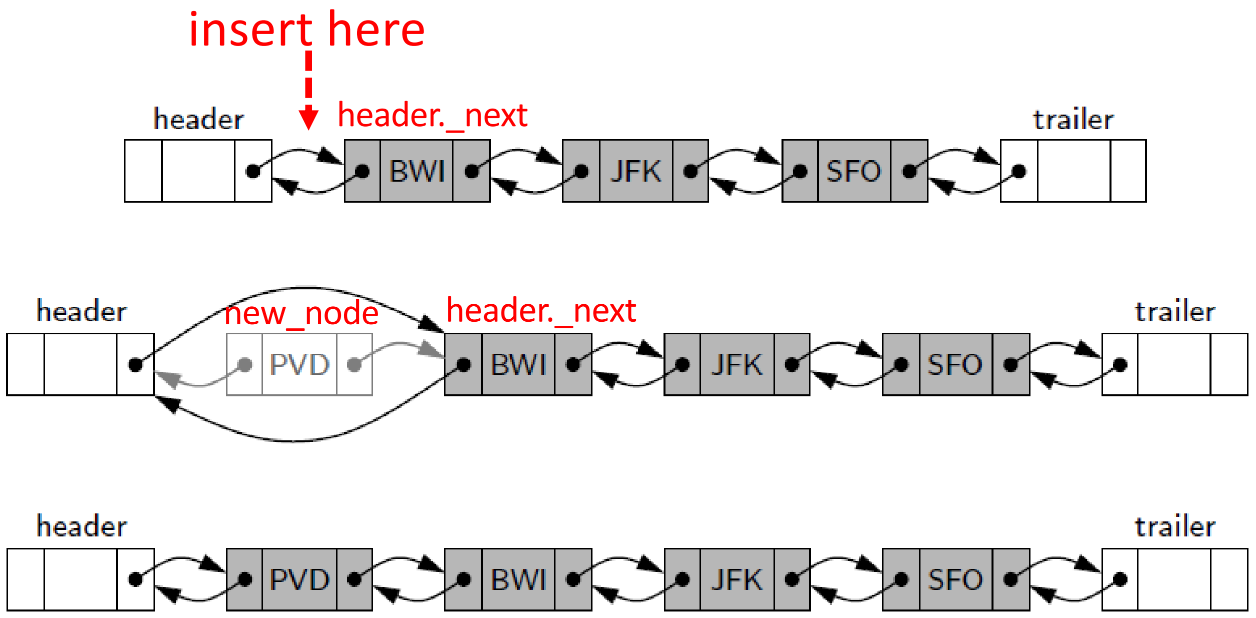
All insertions can be done in the same manner, a new node will always be inserted between a pair of existing nodes.



```
# Insert node between predecessor and successor
  def insert_between(self,item,predecessor,successor):
     newNode = DlinkNode(item,predecessor,successor)
     predecessor._next = newNode
     successor._prev = newNode
```

insert between current_node._prev and current_node

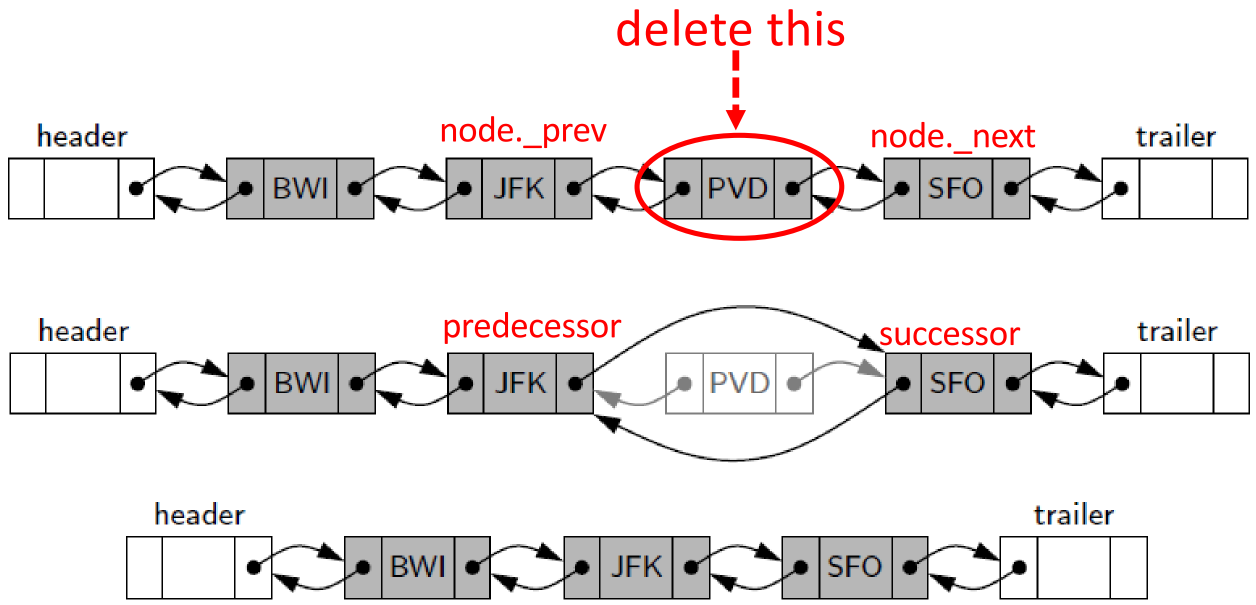
Prepending a node to a doubly linked list



```
# Insert node between predecessor and successor
  def insert_between(self,item,predecessor,successor):
     newNode = DlinkNode(item,predecessor,successor)
     predecessor._next = newNode
     successor._prev = newNode
```

insert between header and header._next

Deleting a node from a doubly linked list



```
def delete_node(self,node):
    predecessor = node._prev
    successor = node._next
    predecessor._next = successor
    successor._prev = predecessor
```

To release node:
node._prev = node._next = node._item = None

Implementing a Deque with a Doubly Linked list

With an implementation based upon a doubly linked list, we can achieve all deque operation in *worst-case* $O(1)$ time.

Deque	Dllist
AddFirst( )	insert_between(header , header._next)
AddRear( )	insert_between(trailer._prev , trailer)
DeleteFirst( )	delete_node(header._next)
DeleteRear( )	delete_node(trailer._prev)