Al-Mustaqbal University
College of Healthcare and Medical Techniques
Intelligent Medical System Department

جامعة المستقبل
AL MUSTAQBAL UNIVERSITY

قسم الانظمة الطبية الذكية
Intelligent Medical Systems Department

Subject: Data Structure
Class: Second
Lecturer: Asst. Prof. Mehdi Ebady Manaa

Lecture: (7)
Linked Lists

Linked Lists

A linked list, in its simplest form, is a collection of **nodes** that **together form a linear ordering**.

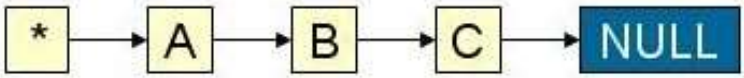
Linked Lists are a very common way of storing arrays of data. The major benefit of linked lists is that you do not specify a fixed size for your list. The more elements you add to the chain, the bigger the chain gets.

There is **more than one type of a linked list**, we'll stick to **singly linked lists** (the simplest one). If for example you want a **doubly linked list** instead, very few simple modifications will give you what you're looking for. Many data structures (e.g. Stacks, Queues, Binary Trees) are often implemented using the concept of linked lists. Some different types of linked lists are shown below:

A few basic types of Linked Lists

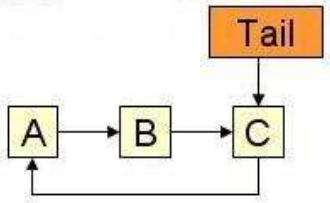
Singly Linked List

Root node links one way through all the nodes. Last node links to null.



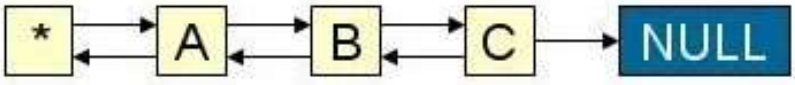
Circular Linked List

Circular linked lists have a reference to one node which is the tail node and all the nodes are linked together in **one direction** forming a circle. The benefit of using circular lists is that appending to the end can be done very guickly.

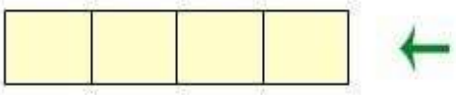


Doubly Linked List

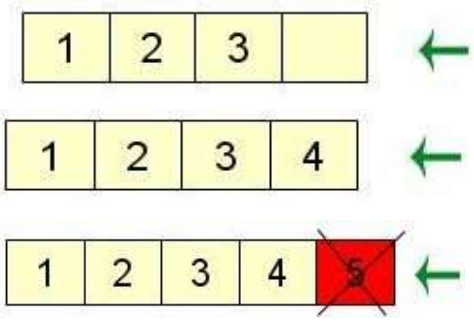
Every node stores a reference to its previous node as well as its next. This is good if you need to move back by a few nodes and don't want to run from the beginning of the list.



There's a diagram to help you realize the main **disadvantage of arrays** but not Linked Lists

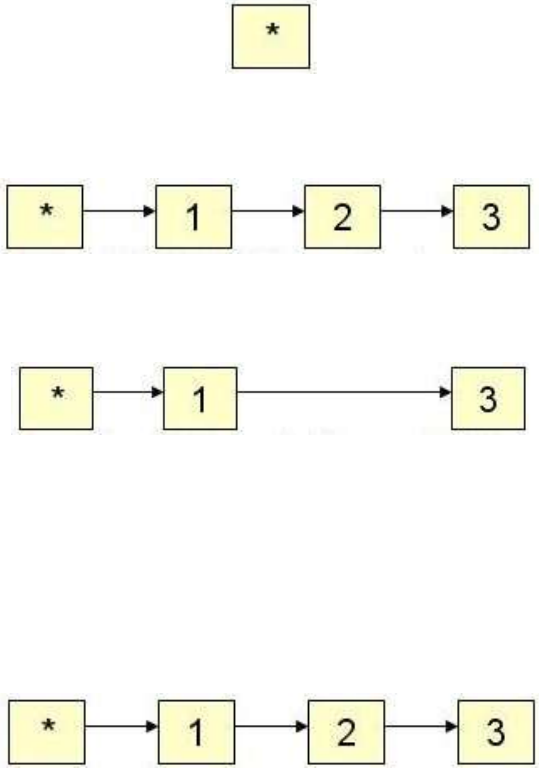


Create an empty integer array.



Fill it partially with some data.The array component without a number indicates allocated but unused space. This is space you could have used for something better.
Add another element. We now have a full array to which we can not add any more elements. We can delete or replace, but we can not add.
lf the array is full, you can not add more elements, because arrays have a fixed size. Linked Lists do not.

Another drawback of arrays is that if you delete an element from the **middle** and want no holes in your array (e.g. (1, 2, 4, null) instead of (1, 2, null, 4)), you will need to shift everything after the deleted element down in O(n) time. If you're trying to add an element somewhere other than the very end of an array, you will need to **shift** some elements towards the end by one (also O(n) time) to make room for the new element, and if you're writing an application which needs to perform well and needs to do these operations often, you should consider using **Linked Lists** instead. This should help you understand Linked Lists:



This is an empty linked list look like. The ***** is an empty node (each element in a linked list called a **node**) which has its next-node reference set to the first node in the list. Since we don't have a first node, its next –node is a null pointer
This is a Linked List with **three nodess**. Each node points to the **next node** in the chain. As mentioned above, ***** is an empty node with a reference to the first node. **[3]** is the last node in the chain with **next ==null**
Here we have delete node **[2]**, so node **[1]** (previosly pointing to **[2]**) now points to **[3]**. If we didn't change the refernce, node **[3]** and any nodes behind it would have no references from your program and get lost. If this happens and you're using **C or C++**, you have a memory leak. If you're using **Java**, the nodes would get automatically garbage collected. Either way, make sure you update the refernces!
For whatever reasons, we've decided to add node **[2]** back nodes **[1]** and **[3]**. The reference from **[1]** is set to **[2]**, and the refence from **[2]** is set to the old reference of **[1]**, which is **[3]**

Pointers

In Python, like in Java, there are no explicit pointer types. Python uses reference semantics, which means **that all variable assignments, method arguments, and elements in data structures (e.g.,**

lists) are handled as references to objects. References and pointers are conceptually similar, but in Python, you don't perform pointer arithmetic, and it's not explicitly necessary because Python manages the memory for you.

In Python, you can easily create data structures like linked lists using reference semantics. For example :

```
class Link:
   def __init__(self, value, next_link):
      self.value = value
      self.next = next_link

if __name__ == "__main:"__
   head = None

   for i in range:(11 ,1)
      head = Link(i, head)

   p = head
   while p is not None:
      print(p.value)
      p = p.next
```

When working with linked data structures, like linked lists, you don't need to explicitly declare references as you do in some languages like Java. You can simply create objects and link them together by assigning references to one another. For example:

```
class Link:
def __init__(self, id, dd):
self.iData = id
self.dData = dd
self.next = None
```

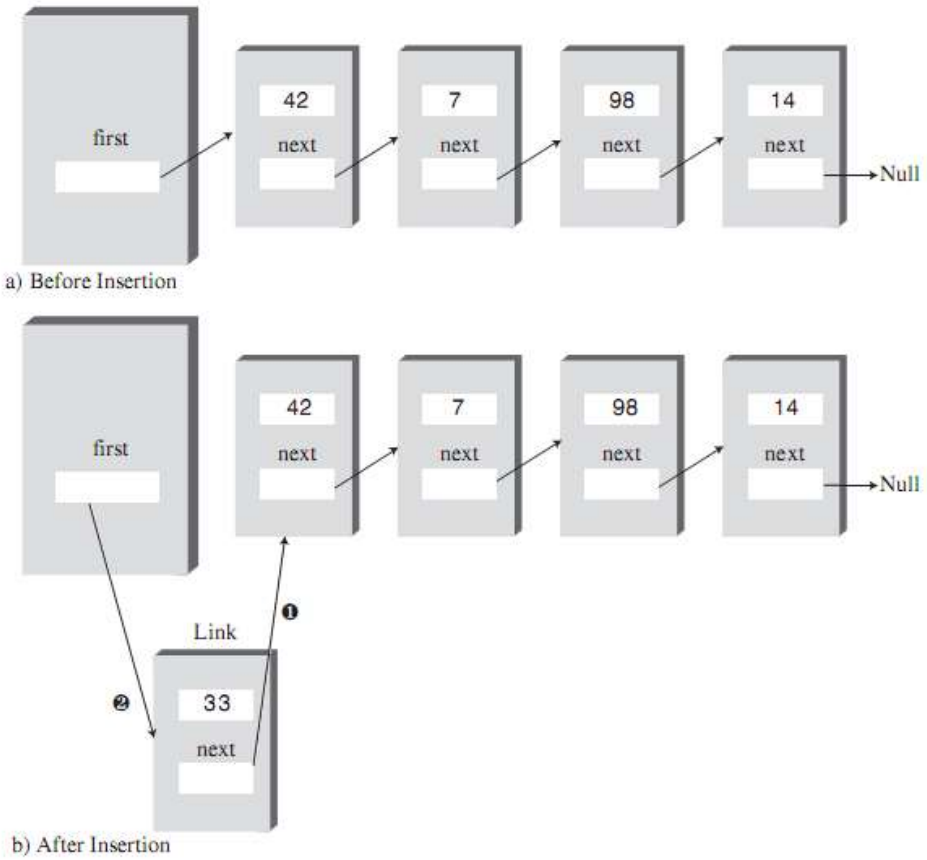
In this Python class, **self.next** is a reference to the next **Link** object in the list. You can create instances of this class and link them together by updating the **self.next** reference. This is how you build linked data structures in Python.
Python uses references implicitly, and you work with objects and references directly, without the need for explicit pointers as seen in some other languages.

The insertFirst() Method

The insertFirst() method of LinkList inserts a new link at the **beginning of the list**.
To insert the new link, we need only set the next field in the newly created link to point to the old first link and then change first so it points to the newly created link. This situation is shown in Figure below. The insertion it can be done in two steps:

1. Update the **next** link of a new node, to point to the **current head node**.

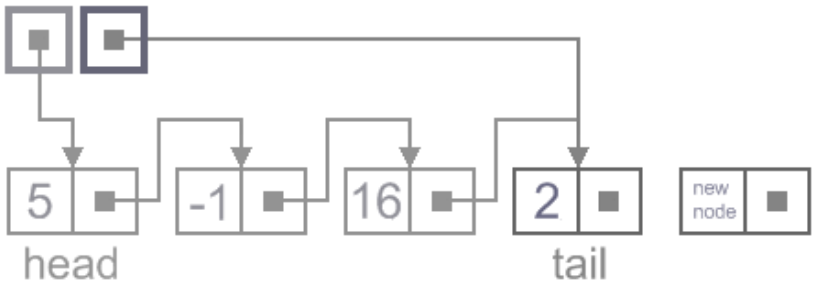
2. Update **head** link to point to the **new node**.



Inserting a new link

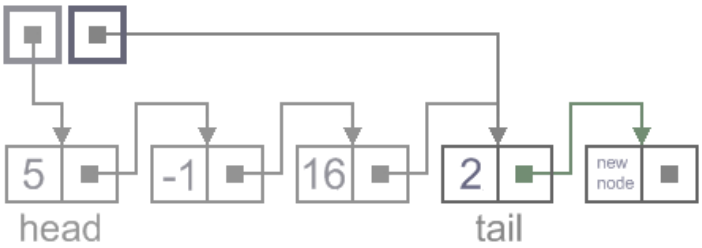
```
def insert_first(self, id, dd):
    # Create a new Link
    new_link = Link(id, dd)
    # Link the newLink to the old first element
    new_link.next = self.first
    # Update the first reference to the newLink
    self.first = new_link
```

Add last

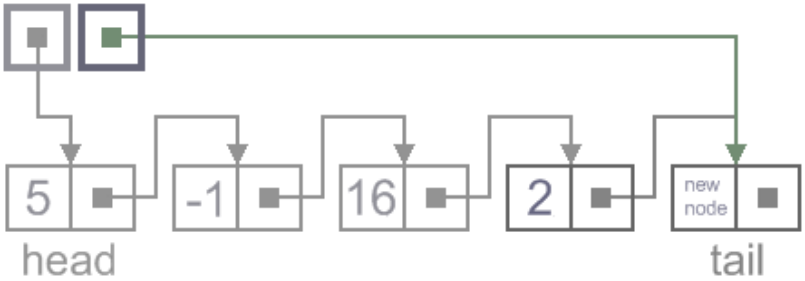


It can be done in two steps:

1. Update the **next link** of the current tail node, to point to the new node.



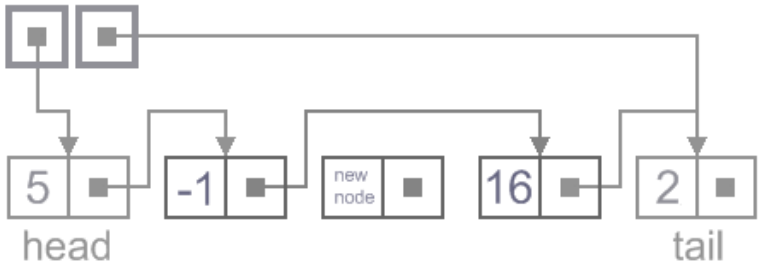
2. Update tail link to point to the new node.




```
def add_last(self, new_node):
    if new_node is None:
        return
    else:
        new_node.next = None
        if self.head is None:
            self.head = new_node
            self.tail = new_node
        else:
            self.tail.next = new_node
```

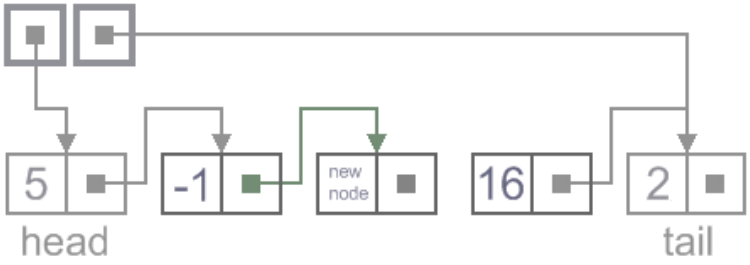
```
        self.tail = new_node
        }    }
```

Inserted Between Two Nodes

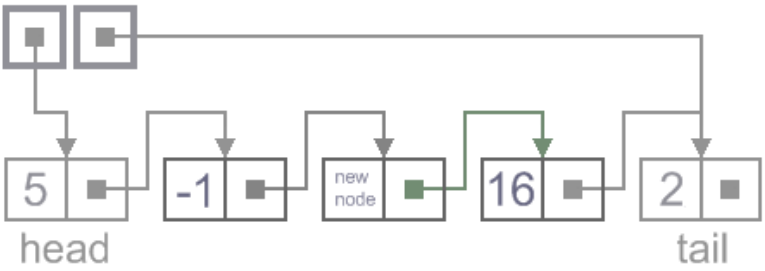


Such an insert can be done in two steps:

1. Update link of the **"previous"** node, to point to the **new node**.



2. Update link of the **new node**, to point to the **"next"** node.




```
def insert_after(self, previous, new_node):
if new_node is None:
   return
else:
   if previous is None:
      self.insert_first(new_node)
   elif previous == self.tail:
      self.add_last(new_node)
   else:
      new_node.next = previous.next
      previous.next = new_node
```

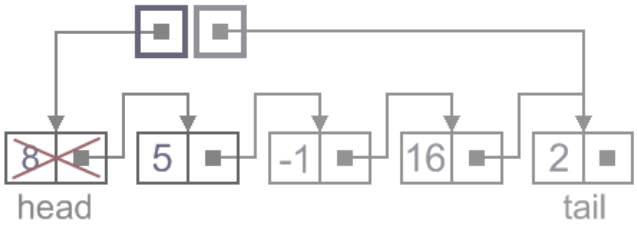
Singly-linked list, Removal (deletion) operation.

There are **four cases**, which can occur while **removing** the node. These cases are similar to the cases in add operation. We have the same four situations, but the order of algorithm actions is opposite. Notice, that removal algorithm includes the **disposal** of the deleted node, which may be unnecessary in languages with automatic garbage collection (i.e., Java).

List has only one node

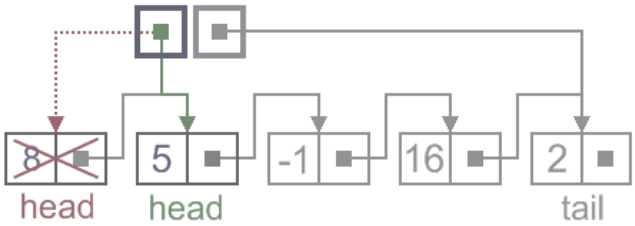
When list has only one node, which is indicated by the condition, that the **head points** to the **same node as the tail**, the removal is quite simple. Algorithm disposes the node, pointed by head (or tail) and **sets both head and tail to NULL**.

Remove first

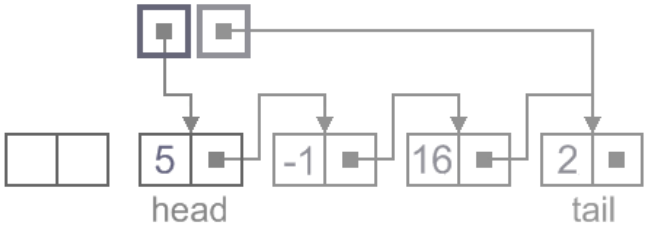


In this case, first node (current head node) is removed from the list.
It can be done in two steps:

1. Update **head link** to point to the node, next to the head.



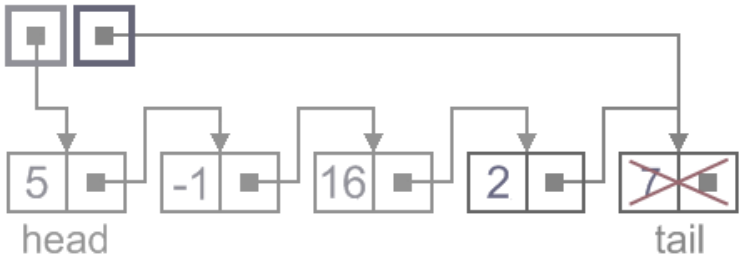
2. Dispose removed node.



```
def remove_first(self):
    if self.head is None:
        return
    else:
        if self.head == self.tail:
            self.head = None
            self.tail = None
        else:
            self.head = self.head.next
```

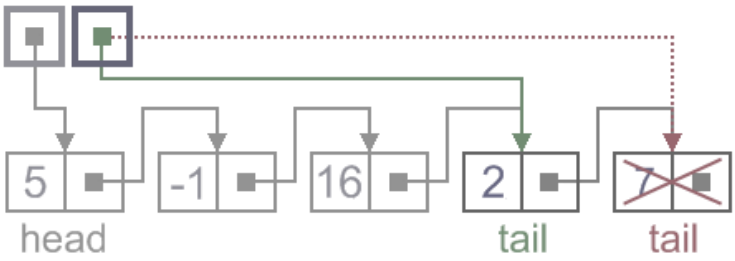
Remove_last

In this case, last node (**current tail** node) is removed from the list. This operation is a bit more tricky, than removing the first node, **because algorithm should find a node**, which is previous to the tail first.

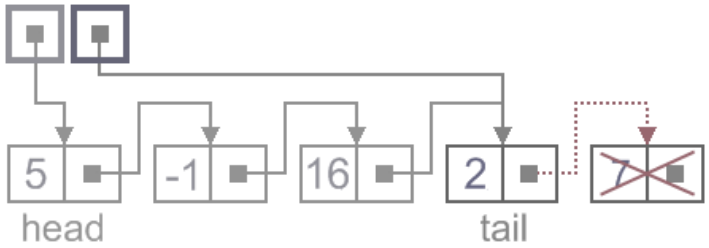


It can be done in **three steps**:

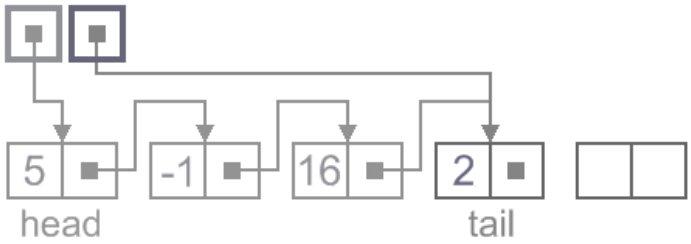
1. Update **tail link** to point to the node, before the tail. In order to find it, list should be **traversed first, beginning from the head.**



2. Set **next link** of the new tail to **NULL**.



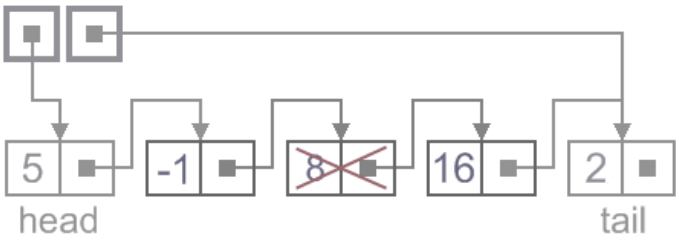
3. Dispose removed node.



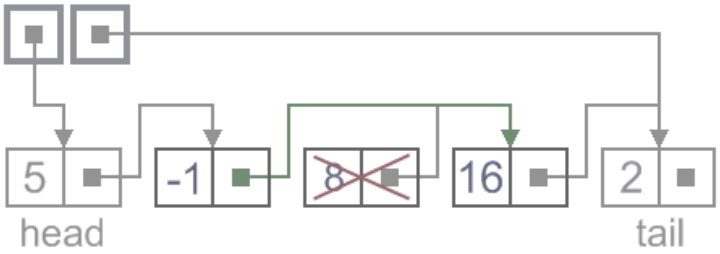

```
def remove_last(self):
    if self.tail is None:
        return
    else:
        if self.head == self.tail:
            self.head = None
            self.tail = None
        else:
            previous_to_tail = self.head
            while previous_to_tail.next != self.tail:
                previous_to_tail = previous_to_tail.next
            self.tail = previous_to_tail
            self.tail.next = None
```

Remove Next

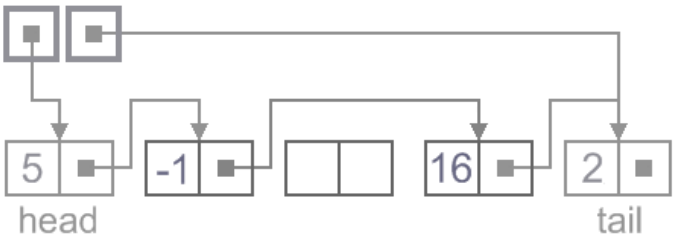
Such a removal can be done in **two steps**:



1. Update **next link** of the **previous node**, to point to the **next node**, relative to the removed node.



2. Dispose removed node.




```
def remove_next(self, previous):
    if previous is None:
        self.remove_first()
    elif previous.next == self.tail:
        self.tail = previous  # Remove last
        self.tail.next = None
    elif previous == self.tail:
        return
    else:
        previous.next = previous.next.next
```

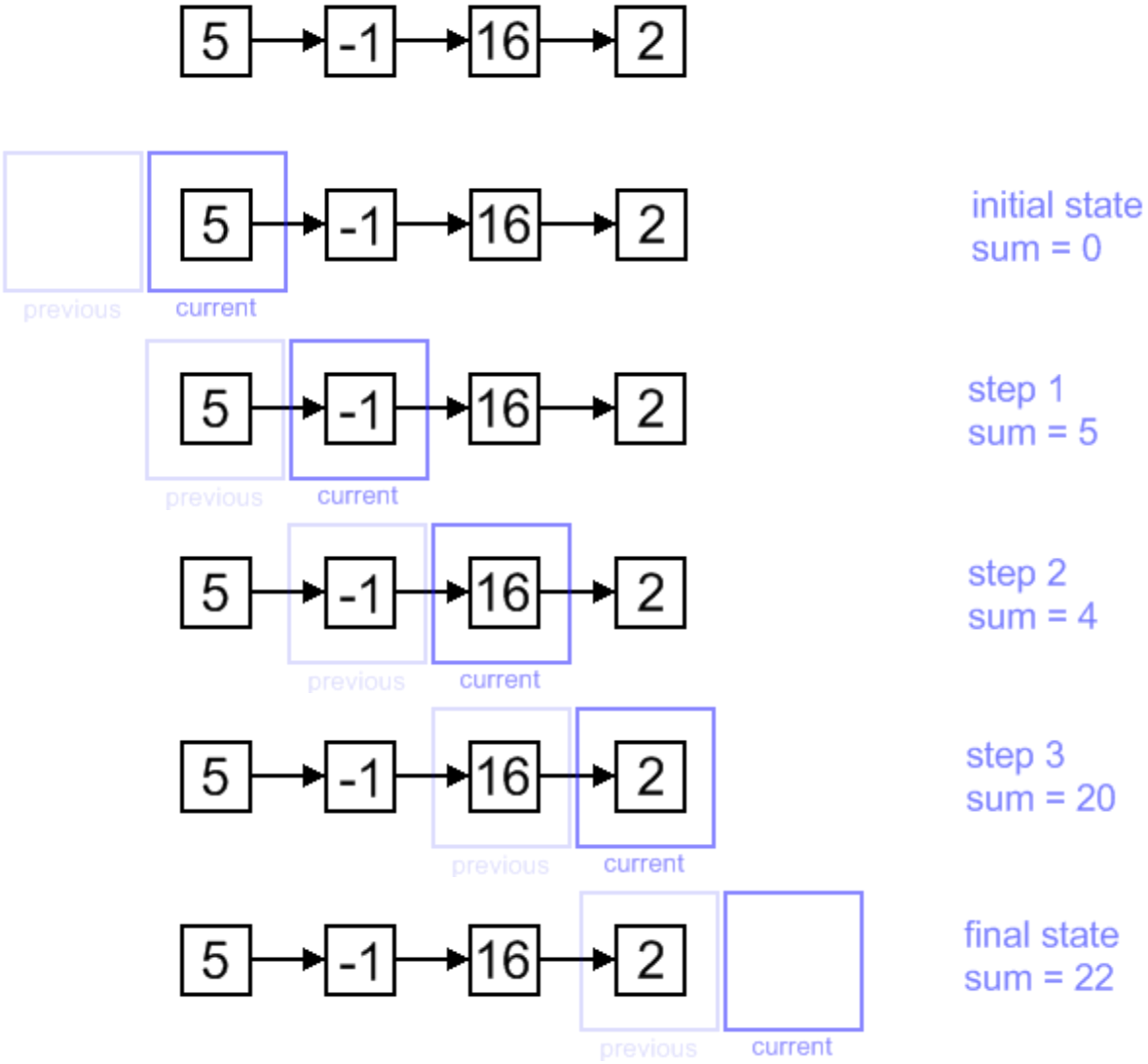
Traversal algorithm

Beginning from the head,

1. check, if the end of a list hasn't been reached yet;
2. do some actions with the current node, which is specific for particular algorithm;
3. current node becomes previous and next node becomes current. Go to the step 1.

Example

As for example, let us see an example of summing up values in a singly-linked list.



For some algorithms tracking the previous node is essential, but for some, like an example, it's unnecessary. We show a common case here and concrete algorithm can be adjusted to meet its individual requirements.

```
class LinkedList:
) ... #    other methods and class definitions(

    def traverse(self):
        sum = 0
        current = self.head
        previous = None
        while current is not None:
```

```
        sum += current.value
        previous = current
        current = current.next
    return sum
```

H.W : Finding and Deleting Specified Links

This example program search a linked list for a data item with a specified key value and to delete an item with a specified key value.

A Stack Implemented by a Linked List

When we created a stack we used an ordinary Java array to hold the stack's data. The stack's push() and pop() operations were actually carried out by array operations such as

arr[++top] = data;
and
data = arr[top--];
which insert data into, and take it out of, an array.

We can also use a **linked list** to hold a stack's data. In this case the push() and pop() operations would be carried out by operations like

theList.insertFirst(data)
and
data = theList.deleteFirst()

The user of the stack class calls push() and pop() to insert and delete items without knowing, or needing to know, whether the stack is implemented as an array or as a linked list.

```
class Link:
    def __init__(self, dData):
        self.dData = dData
        self.next = None

class LinkList:
    def __init__(self):
        self.first = None

    def is_empty(self):
        return self.first is None

    def insert_first(self, dd):
        new_link = Link(dd)
        new_link.next = self.first
        self.first = new_link
```

```
    def delete_first(self):
        if not self.is_empty():
            temp = self.first
            self.first = self.first.next
            return temp.dData
        else:
            return None

    def display_list(self):
        current = self.first
        while current is not None:
            print(current.dData, end=" ")
            current = current.next
        print()

class LinkStack:
    def __init__(self):
        self.the_list = LinkList()

    def push(self, j):
        self.the_list.insert_first(j)

    def pop(self):
        return self.the_list.delete_first()

    def is_empty(self):
        return self.the_list.is_empty()

    def display_stack(self):
        print("Stack (top-->bottom): ", end="")
        self.the_list.display_list()

class FirstLastList:
    def __init__(self):
        self.first = None
        self.last = None

    def is_empty(self):
        return self.first is None

    def insert_last(self, dd):
        new_link = Link(dd)
        if self.is_empty():
            self.first = new_link
        else:
            self.last.next = new_link
        self.last = new_link

    def delete_first(self):
        if not self.is_empty():
            temp = self.first
            self.first = self.first.next
```

```
            if self.first is None:
                self.last = None
            return temp.dData
        else:
            return None

    def display_list(self):
        current = self.first
        while current is not None:
            print(current.dData, end=" ")
            current = current.next
        print()

class LinkQueue:
    def __init__(self):
        self.the_list = FirstLastList()

    def is_empty(self):
        return self.the_list.is_empty()

    def insert(self, j):
        self.the_list.insert_last(j)

    def remove(self):
        return self.the_list.delete_first()

    def display_queue(self):
        self.the_list.display_list()

# Example usage of the classes and methods:

# Creating a stack and using it
stack = LinkStack()
stack.push(1)
stack.push(2)
stack.push(3)
stack.display_stack()
stack.pop()
stack.display_stack()

# Creating a queue and using it
queue = LinkQueue()
queue.insert(1)
queue.insert(2)
queue.insert(3)
queue.display_queue()
queue.remove()
queue.display_queue()
```

Output
Stack (top-->bottom): 3 2 1
Stack (top-->bottom): 2 1
1 2 3
2 3