CMC UNIVERSITY

CHAPTER 2.
BASIC DATA STRUCTURES

CMC UNIVERSITY

Outline

- ✓ Arrays and Lists
- ✓ Stacks
- ✓ Queues
- ✓ Hash Tables

CMC UNIVERSITY

ARRAYS AND LISTS

CMC UNIVERSITY

The basic concepts

Arrays

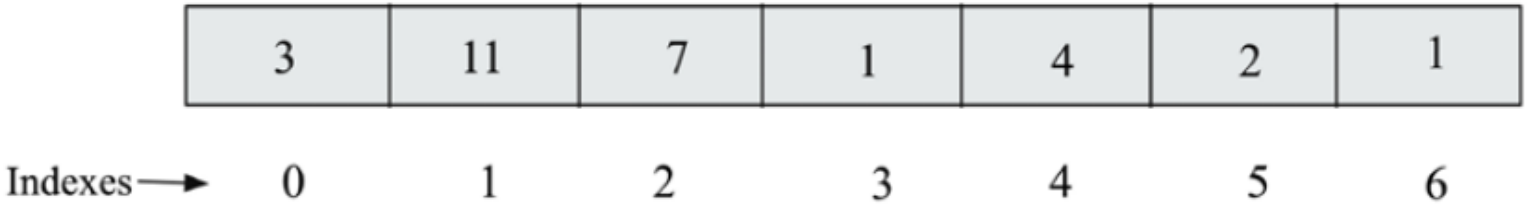
- An array is an ordered set of a fixed number of elements of the same type, each data element in the array is stored in contiguous memory locations.
- Storing multiple data values of the same type makes it easier and faster to compute the position of any element in the array using offset and base address.
- The term base address refers to the address of memory location where the first element is stored, and offset refers to an integer indicating the displacement between the first element and a given element.

CMC UNIVERSITY

The basic concepts

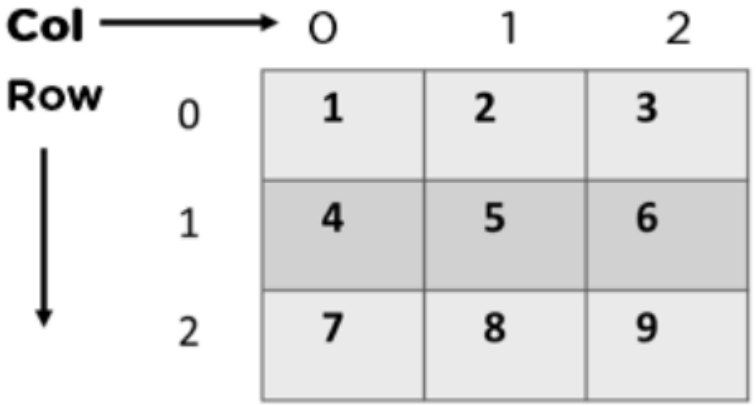
There are majorly two types of arrays, they are:

- One-Dimensional Arrays:

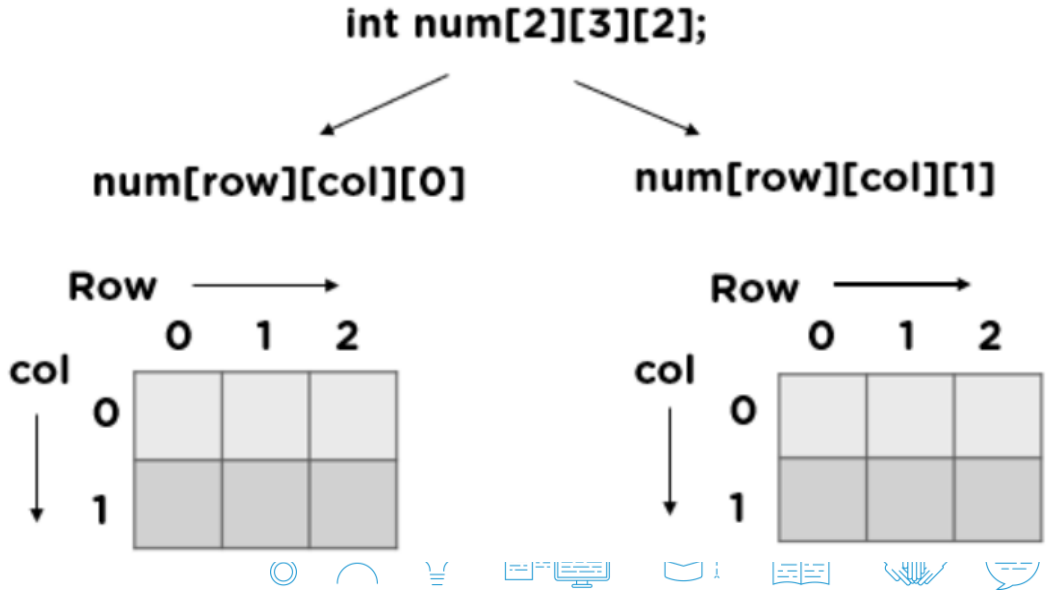


- Multi-Dimensional Arrays:

Two-Dimensional Arrays



Three-Dimensional Arrays



CMC UNIVERSITY

The basic concepts

- Can create arrays, insert an element into the array, remove an element from the array, search for elements in the array, updating an element of array, etc.
- In addition to the value, an element of the array is also characterized by an index representing the order of that element in the array.
- Example:
  - Vector is a one-dimensional array, each element a_i corresponds to an index i.
  - Matrix is a two-dimensional array, each element a_{ij} corresponds to two indices i and j.
  - Similarly, one also extends 3-dimensional arrays, 4-dimensional arrays, ..., n-dimensional arrays.

CMC UNIVERSITY

The basic concepts

Lists

- A *list* is an ordered set of a dynamic number of elements. A list in which the neighbor relationship between the elements is displayed is called a *linear list*.
- Example: the collection of people who come to buy train tickets gives us an image of a list. They will buy tickets in order.

(Vector is a special case of a linear list, which is an image of a linear list at a time)

- A linear list is an empty list (no elements) or of the form (a_1, a_2, ..., a_n), where a_i is the atomic data, a_1 is the first element and a_n is the last element. For each element a_i, a_{i+1} is the next element, a_{i-1} is the prior element, n is the length (size) of the list, and the value n can change.
- Each element in the list is usually a record (consisting of one or more fields).

CMC UNIVERSITY

The basic concepts

There are some common operations:

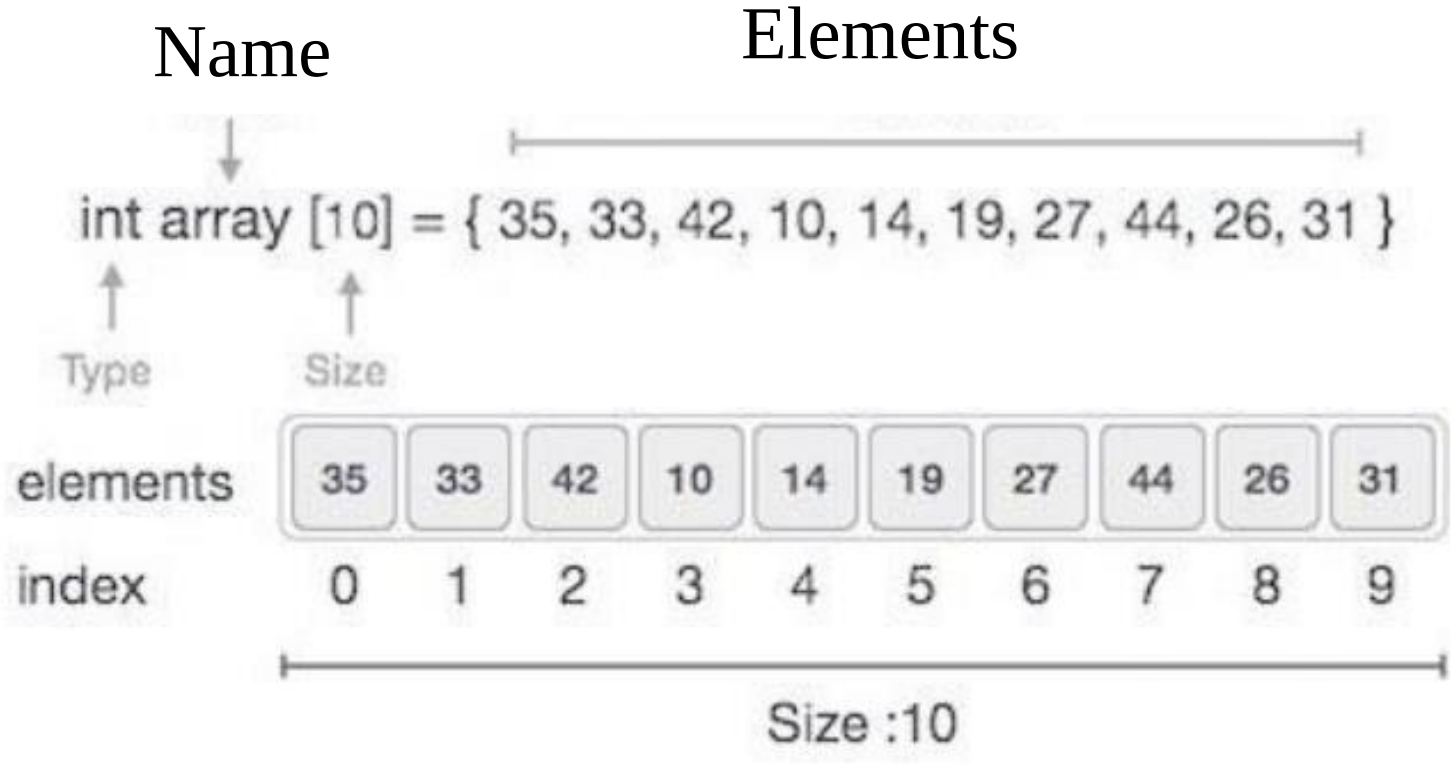
- Accessing Values
- Insert an element into lists
- Delete an element from lists
- Concatenate two or more lists
- Split a list into multiple lists
- Copy a list
- Update a list

Array Representation

Arrays can be declared in various ways in different languages

Example:

- Index starts with 0.
- Array length is 10, which means it can store 10 elements.
- Each element can be accessed via its index. For example, we can fetch an element at index 6 as 9.



CMC UNIVERSITY

Basic Operations

The basic operations supported by an array are as stated below −

- ***Traverse*** − print all the array elements one by one.
- ***Insertion*** − Adds an element at the given index.
- ***Deletion*** − Deletes an element at the given index.
- ***Search*** − Searches an element using the given index or by the value.
- ***Update*** − Updates an element at the given index.

CMC UNIVERSITY

Basic Operations

Array is created in Python by importing array module to the python program. Then, the array is declared as shown below −

```
from array import *
arrayName = array(typecode, [Initializers])
```

Typecode are the codes that are used to define the type of value the array will hold. Some common typecodes used are as follows

Typecode	Value
B	Represents signed integer of size 1 byte
B	Represents unsigned integer of size 1 byte
C	Represents character of size 1 byte
i	Represents signed integer of size 2 bytes
I	Represents unsigned integer of size 2 bytes
F	Represents floating point of size 4 bytes
d	Represents floating point of size 8 bytes

CMC UNIVERSITY

Basic Operations

Example:
Creates an array named a

```
from array import *
a = array('i', [10,20,30,40,50,60])
for x in a:
   print(x)
```

Example:
Access each element of an array using the index of the element.

```
from array import *
a = array('i', [10,20,30,40,50,60])
print (a[0])
print (a[2])
```

Insert operation is to insert one or more data elements into an array.
A new element can be added at the beginning, end, or any given index of array

```
from array import *
a = array('i', [10,20,30,40,50,60])
a.insert(1,80)
for x in a:
   print(x)
```

CMC UNIVERSITY

Basic Operations

Deletion refers to removing an existing element from the array and re-organizing all elements of an array

```
from array import *
a = array('i', [10,20,30,40,50,60])
a.remove(40)
for x in a:
    print(x)
```

Search for an array element based on its value or its index.

```
from array import *
a = array('i', [10,20,30,40,50,60])
print (a.index(40))
```

Update operation refers to updating an existing element from the array at a given index

```
from array import *
a = array('i', [10,20,30,40,50,60])
a[2] = 80
for x in a:
    print(x)
```

CMC UNIVERSITY

Basic Operations

```
1  from array import *
2  # Enter the size of the array
3  size = int(input("Enter the size of the array: "))
4  # Initialize an empty integer array
5  my_array = array('i')
6  # Enter the value for the array
7  for i in range(size):
8      num = int(input("Enter a value for the element {}: ".format(
           )))
9      my_array.append(num)
10 # Print the created array
11 print("The created array:")
12 for x in my_array:
13     print(x)
```

CMC UNIVERSITY

Basic Operations

- To store, traverse, and access array elements is very fast as compared to lists since elements can be accessed randomly using their index position.
- If the data to be stored in the array is large and the system has low memory, the array data structure will not be a good choice to store the data because it is difficult to allot a large block of memory locations. The array data structure has further limitations in that it has a static size that has to be declared at the time of creation

CMC UNIVERSITY

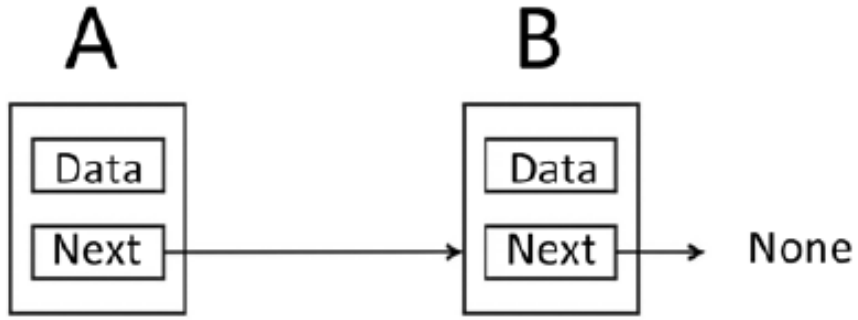
Exercises

- Create an integer array with the number of elements entered by the user.
- Inserts an element with a specified value and position into the array.
- Remove an element at the specified position from the array
- Remove an element with a specified value from the array
- Specifies the position of the element with the specified value in the array

CMC UNIVERSITY

Linked lists

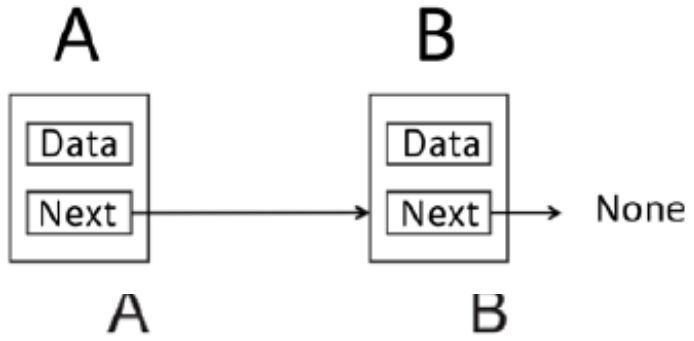
- The use of pointers or links to organize a linear list is called a linked list
- A linked list is a data structure where the data elements are stored in a linear order. Linked lists provide efficient storage of data in linear order through pointer structures. Pointers are used to store the memory address of data items. They store the data and location, and the location stores the position of the next data item in the memory.
- A linear list includes a series of connected nodes, each node stores the data and the address of the next node. For example,



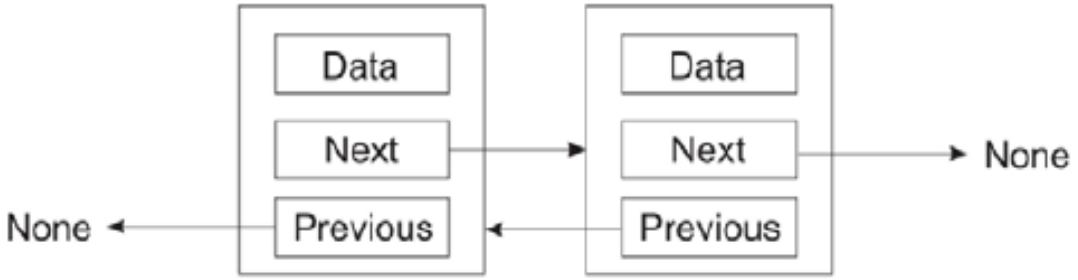
- The length of the list can increase or decrease during the execution of the program.
- Linked lists can be of multiple types: *singly*, *doubly*, and *circular linked list*.

Types of Linked list

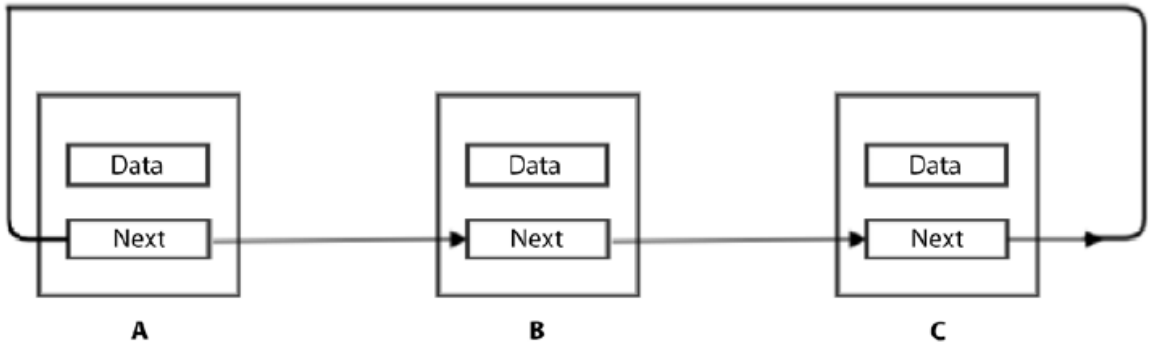
Singly Linked List:
Each node has data and a pointer to the next node.

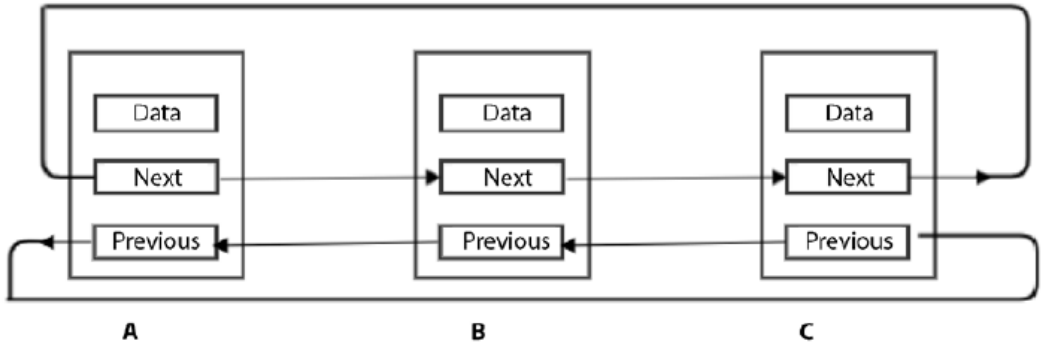


Doubly Linked List: add a pointer to the previous node in a doubly-linked list → forward or backward



Circular linked list: is a variation of a linked list in which the last element is linked to the first element. This forms a circular loop.





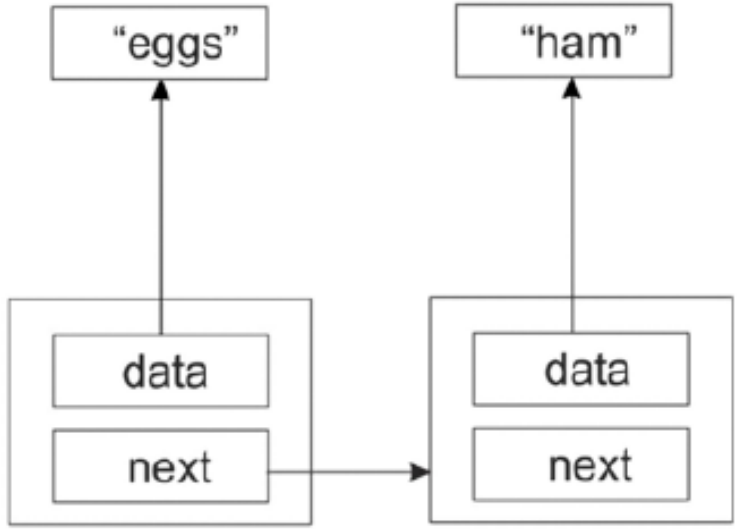
Representing a list with a linked pointer

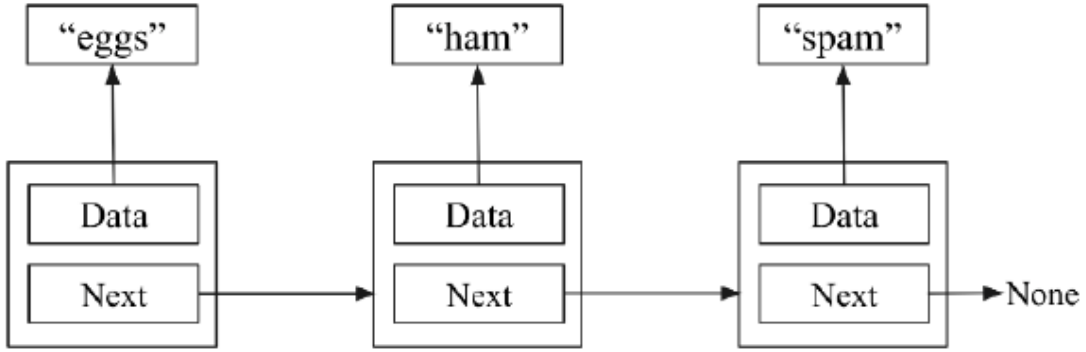
- A node is a key component of several data structures such as linked lists. A node is a container of data, together with one or more links to other nodes where a link is a pointer.

Example:

- Create a linked list of two nodes that contains data (for example, strings). We first declare the variable that stores the data along with pointers that point to the next variable. The first node has a pointer to the string (*eggs*), and another node pointing to the *ham* string

- Add a new third node to this existing linked list that stores *spam* as a data value





CMC UNIVERSITY

Representing a list with a linked pointer

So, we have created three nodes—one containing *eggs*, one *ham*, and another *spam*.

The *eggs* node points to the *ham* node, which in turn points to the *spam* node. The spam node is the last element in the list, it points to nothing. In Python, we will use the special value *None* to denote nothing.

Code snippet: the next pointer is initialized to *None*, meaning that initially, any node that is attached to the list will be independent.

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

If you want to add any other data items to the node class, for example, the node is going to contain customer data, then create a *Customer* class and place all the data there.

CMC UNIVERSITY

Principles of organizing linked lists

To organize the linked list, the following capabilities are required:

- There exist means to divide memory into nodes, where each node can access each field (we only consider the case of nodes and fields of fixed size).
- There exists a mechanism to determine whether a node is in use (called a busy node) or not in use (called an empty node).
- There exists a mechanism as a space repository to provide empty nodes when required to use and reclaim those nodes when no longer needed.

CMC UNIVERSITY

Linked List Operations

There are various linked list operations that allow us to perform different actions on linked lists:

- ***Traversal*** - access each element of the linked list
- ***Insertion*** - adds a new element to the linked list
- ***Deletion*** - removes the existing elements
- ***Search*** - find a node in the linked list
- ***Sort*** - sort the nodes of the linked list

Linked List Operations - Traversal

Creating and traversing

- Create the node class (in the previous section)

Example: create three nodes, n1, n2, and n3, that store three strings:

```
n1 = Node('eggs')
n2 = Node('ham')
n3 = Node('spam')
```

- Create the linked list: link the nodes sequentially

Example: node n1 is pointing to node n2, node n2 is pointing to node n3, and node n3 is the last node, and is pointing to None:

```
n1.next = n2
n2.next = n3
```

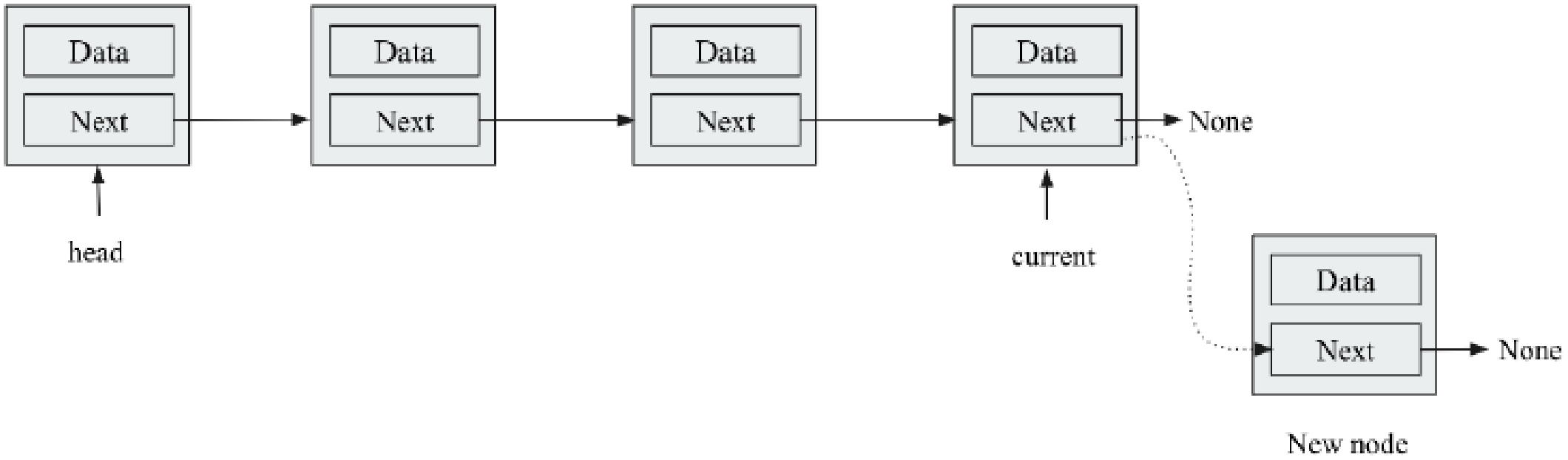
- Travers the linked lists: visit all the nodes of the list, from the starting node to the last node (begins with the first node, displaying the data of the current node, following the pointers, and finally stopping when we reach the last node)

```
current = n1
while current:
    print(current.data)
    current = current.next
```

Linked List Operations - Insertion

Insertion: append items to the list

Appending items to the end of a list



Linked List Operations - Insertion

- We encapsulate data in a node so that it has the next pointer attribute.
- We check if there are any existing nodes in the list (that is, whether self.head points to a Node).
  - If there is None, this means that initially, the list is empty and the new node will be the first node. So, we make the new node the first node of the list;
  - Otherwise, we find the insertion point by traversing the list to the last node and updating the next pointer of the last node to the new node.

```
class SinglyLinkedList:
    def __init__ (self):
        self.head = None
        self.size = 0
def append(self, data):
    # Encapsulate the data in a Node
    node = Node(data)
    if self.head is None:
        self.head = node
    else:
        current = self.head
        while current.next:
            current = current.next
        current.next = node
```

Linked List Operations - Insertion

Example: append three following nodes

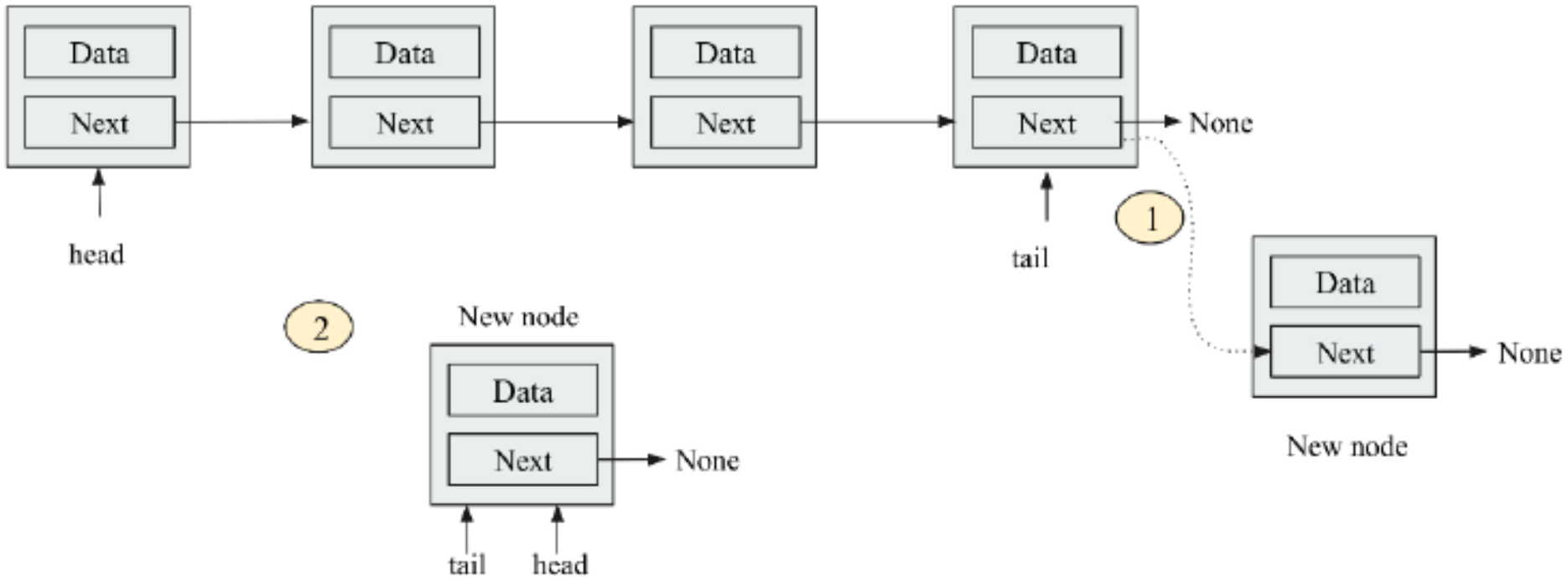
- This implementation is not very efficient when the list is long, and there is a drawback with the append method: we have to traverse the entire list to find the insertion point.

```
words = SinglyLinkedList()
words.append('egg')
words.append('ham')
words.append('spam')
```

```
current = words.head
while current:
    print(current.data)
    current = current.next
```

Linked List Operations - Insertion

- A better implementation of the append method: add a variable in the node that references the last node of the list. The worst-case running time of the append operation can be reduced from O(n) to O(1) using this method. We must ensure that the previous last node points to the new node that is to be appended to the list.



Linked List Operations - Insertion

In the following code, a new node can be appended in the end through a tail pointer by making a link from the last node to the new node.

```
class SinglyLinkedList:
    def __init__ (self):
        self.tail = None
        self.head = None
        self.size = 0
    def append(self, data):
        node = Node(data)
        if self.tail:
            self.tail.next = node
            self.tail = node
        else:
            self.head = node
            self.tail = node
```

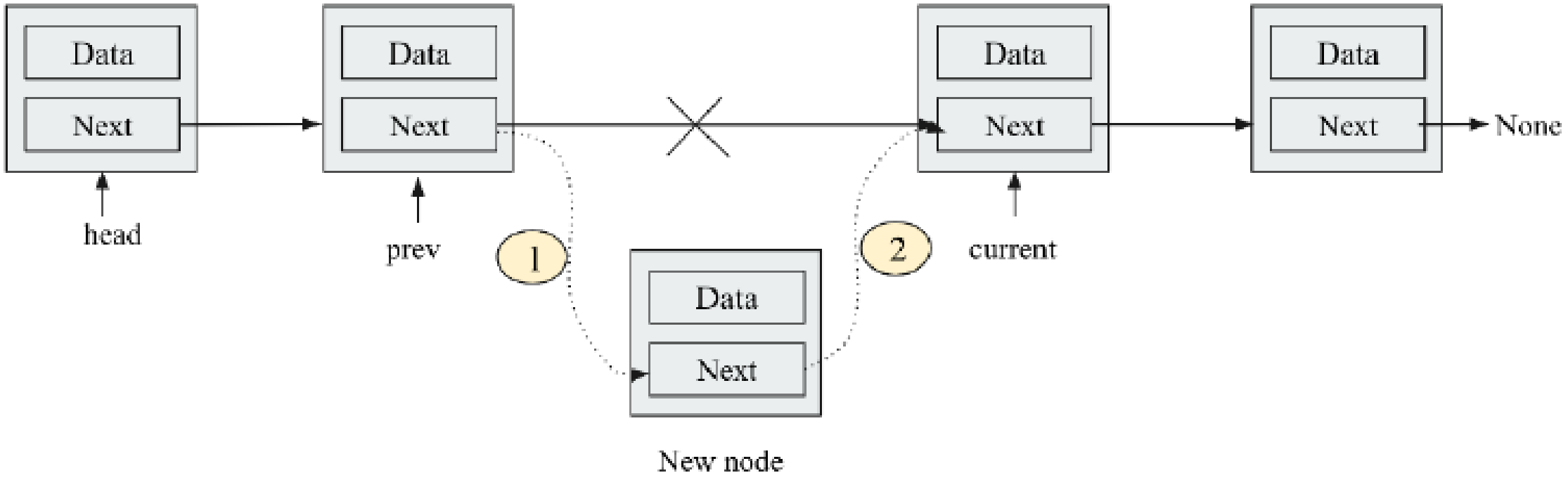
```
words = SinglyLinkedList()
words.append('egg')
words.append('ham')
words.append('spam')

current = words.head
while current:
    print(current.data)
    current = current.next
```

CMC UNIVERSITY

Linked List Operations - Insertion

Appending items at intermediate positions

To append or insert an element in an existing linked list at a given position, firstly, we have to traverse the list to reach the desired position where we want to insert an element. An element can be inserted in between two successive nodes using two pointers (*prev* and *current*).



Linked List Operations - Insertion

When we want to insert a node in between two existing nodes, all we have to do is update two links. The previous node points to the new node, and the new node should point to the successor of the previous node.

```
class SinglyLinkedList:
    def __init__ (self):
        self.tail = None
        self.head = None
        self.size = 0

    def append_at_a_location(self, data, index):
        current = self.head
        prev = self.head
        node = Node(data)
        count = 1
        while current:
            if count == 1:
                node.next = current
                self.head = node
                print(count)
                return
            elif index == index:
                node.next = current
                prev.next = node
                return
```

```
        count += 1
        prev = current
        current = current.next
    if count < index:
        print("The list has less number of elements")
```

```
words = SinglyLinkedList()
words.append('egg')
words.append('ham')
words.append('spam')
current = words.head
while current:
    print(current.data)
    current = current.next
```

```
words.append_at_a_location('new', 2)
current = words.head
while current:
    print(current.data)
    current = current.next
```

Linked List Operations - Insertion

It is important to note that the condition where we may want to insert a new element can change depending upon the requirement, so let's say we want to insert a new element just before an element that has the same data value. In that case, the code to *append_at_a_position* will be as follows:

```
def append_at_a_location(self, data):
    current = self.head
    prev = self.head
    node = Node(data)
    while current:
        if current.data == data:
            node.next = current
            prev.next = node
        prev = current
        current = current.next
```

We can now use the preceding code to insert a new node at an intermediate position:

```
words.append_at_a_location('ham')
current = words.head
while current:
    print(current.data)
    current = current.next
```

CMC UNIVERSITY

Linked List Operations - Insertion

The worst-case time complexity of the insert operation is O(1) when we have an additional pointer that points to the last node.

Otherwise, when we do not have the link to the last node, the time complexity will be O(n) since we have to traverse the list to reach the desired position and in the worst case, we may have to traverse all the n nodes in the list.

CMC UNIVERSITY

Linked List Operations - Search

Search

Searching an element in a list

We may also need to check whether a list contains a given item. This can be implemented using the iter() method, which we have already seen in the previous section while traversing the linked list. Using that, we write the search method as follows

```
def search(self, data):
    for node in self.iter():
        if data == node:
            return True
    return False
```

In the above code, each pass of the loop compares the data to be searched with each data item in the list one by one. If a match is found, True is returned, otherwise False is returned.

```
print(words.search('sspam'))
print(words.search('spam'))
```

Output:

False

True

CMC UNIVERSITY

Linked List Operations - Search

Getting the size of the list

To get the size of the list by counting the number of nodes: traversing the entire list and increasing the counter as we go along.

```
def size(self):
    count = 0
    current = self.head
    while current:
        count += 1
        current = current.next
    return count
```

The above code is very similar to traverse the nodes of the list one by one and increase the count variable. However, list traversal is potentially an expensive operation that we should avoid wherever we can.

An another method in which we can add a size member to the *SinglyLinkedList* class, initializing it to 0 in the constructor

```
class SinglyLinkedList:
    def __init__(self):
        self.head = data
        self.size = 0
```

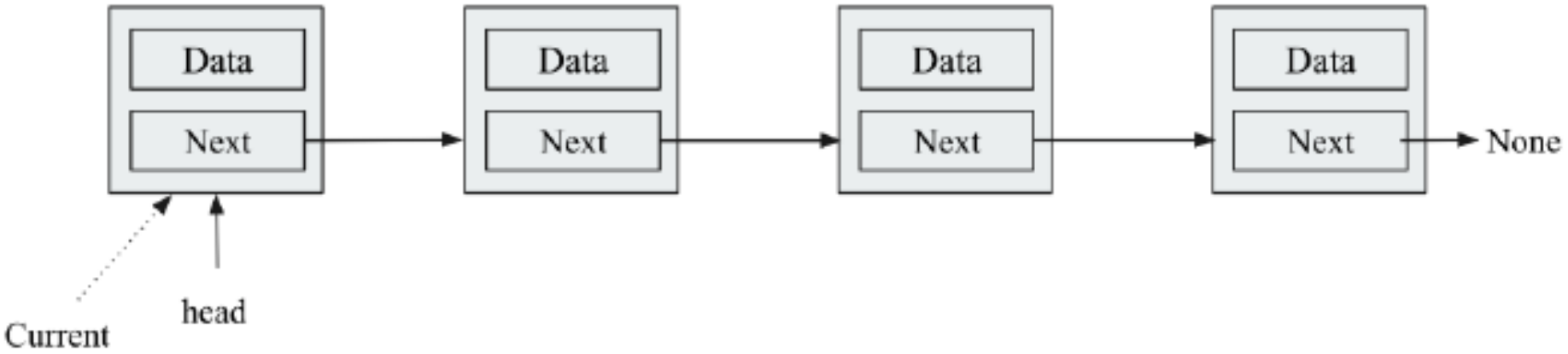
Because we are now only reading the size attribute of the node object, we reduce the worst-case running time from O(n) to O(1).

Linked List Operations - Delete

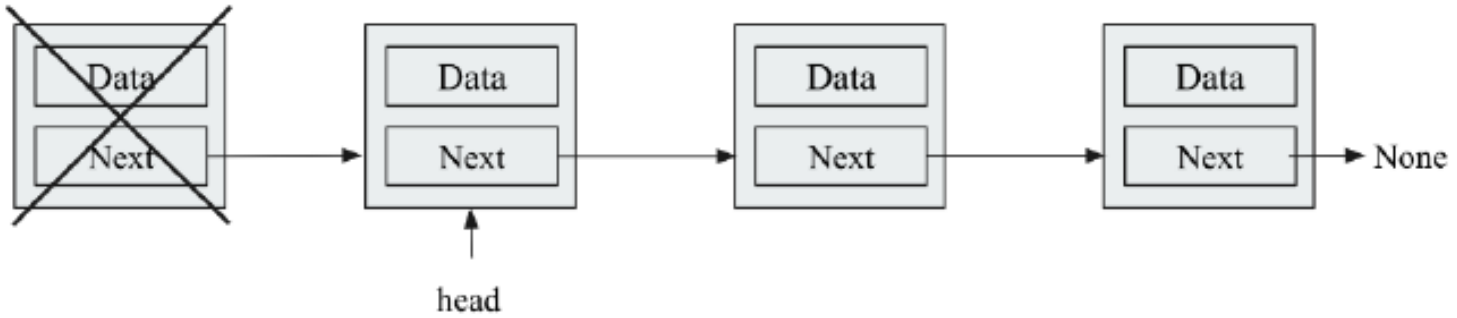
Delete

Deleting the node at the beginning of the singly linked list

-A temporary pointer (current pointer) is created that points to the first node (head node)



Next, the current node pointer is moved to the next node and assigned to the head node. Now, the second node becomes the head node that is pointed to by the head pointer



CMC UNIVERSITY

Linked List Operations - Delete

In this code:

- We initially check if there is no item to delete from the list, and we print the appropriate message.
- Next, if there is some data item in the list, we assign the head pointer to the temporary pointer current as per step 1, and then the head pointer is now pointing to the next node, assuming that we already have a linked list of three data items – *eggs*, *ham*, *spam*

```
def delete_first_node (self):
        current = self.head
         if self.head is None:
                print("No data element to delete")
         elif current == self.head:
                self.head = current.next
```

```
words.delete_first_node()
current = words.head
while current:
    print(current.data)
    current = current.next
```

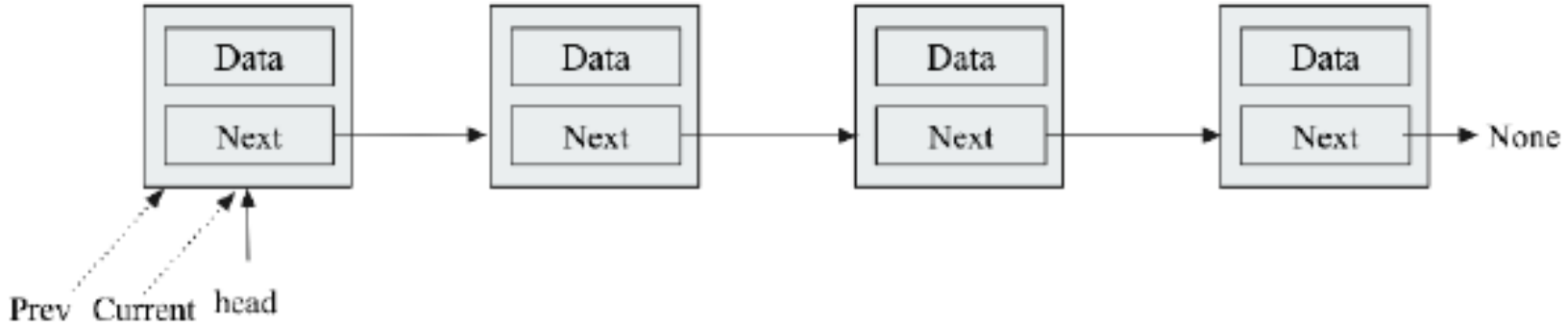
Output:

ham
spam

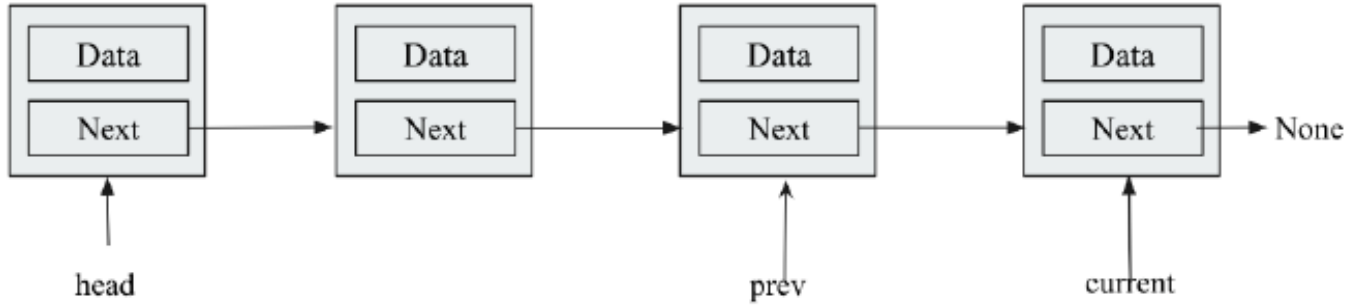
Linked List Operations - Delete

Deleting the node at the end of the singly linked list

- Firstly, we have two pointers, in other words, a *current* pointer that will point to the last node, and a *prev* pointer that will point to the node previous to the last node (second last node). Initially, we will have three pointers (current, prev, and head) pointing to the first node:

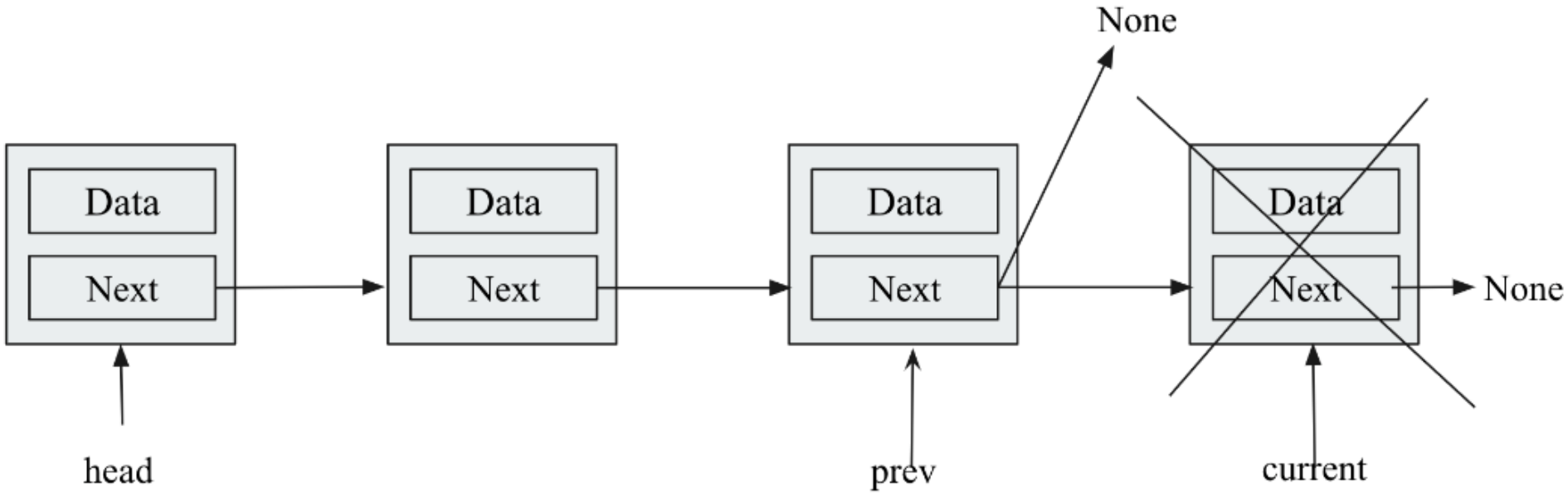


- To reach the last node, we move the *current* and *prev* pointers in such a way that the current pointer should point to the last node and the *prev* pointer should point to the second last node. So, we stop when the *current* pointer reaches the last node.



Linked List Operations - Delete

- Finally, we mark the *prev* pointer to point to the second last node, which is rendered as the last node of the list by pointing this node to *None* :



CMC UNIVERSITY

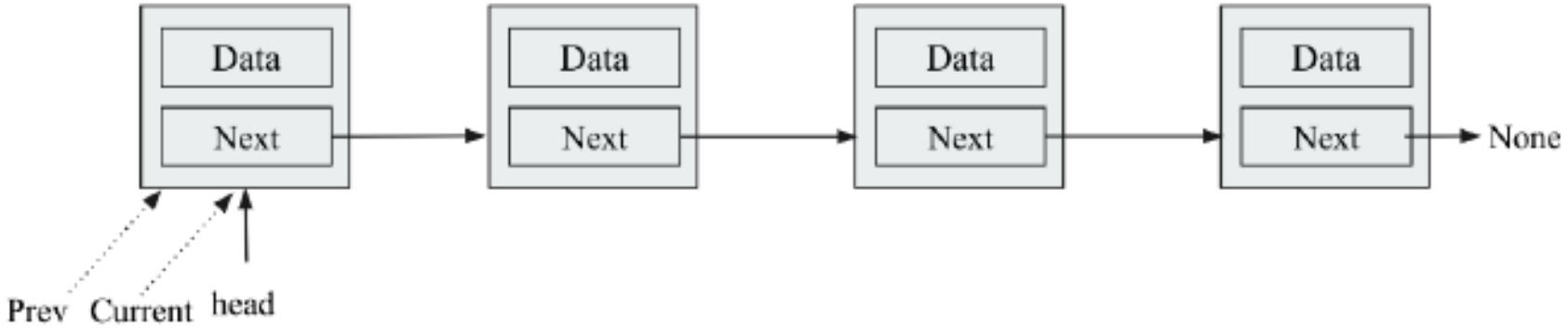
Linked List Operations - Delete

```
def delete_last_node (self):
    current = self.head
    prev = self.head
    while current:
        if current.next is None:
            prev.next = current.next
            self.size -= 1
        prev = current
        current = current.next
```

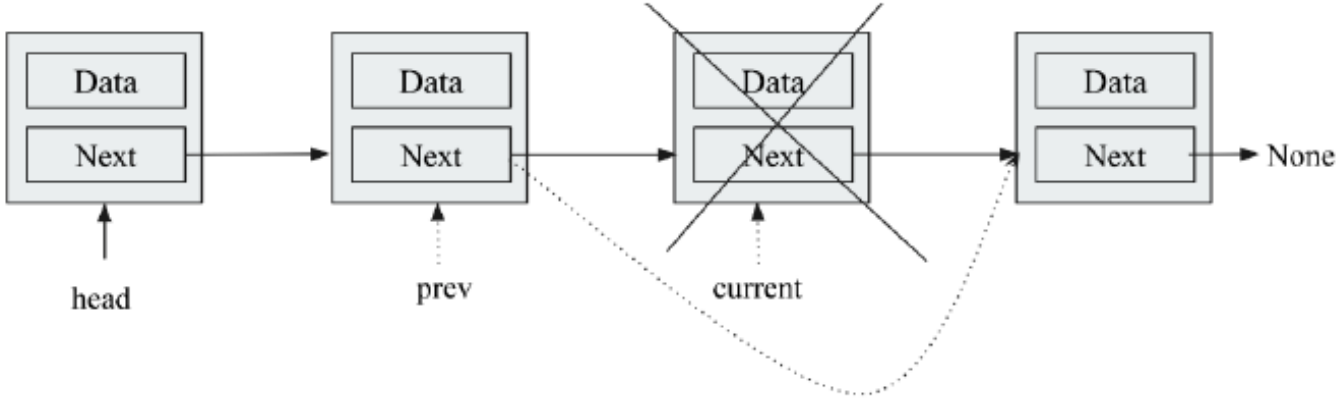
Linked List Operations - Delete

Deleting any intermediate node in a singly linked list

- The initial pointers point to the first node



- Once the node is identified, the prev pointer is updated to delete the node. The node to be deleted is shown along with the link to those to be updated

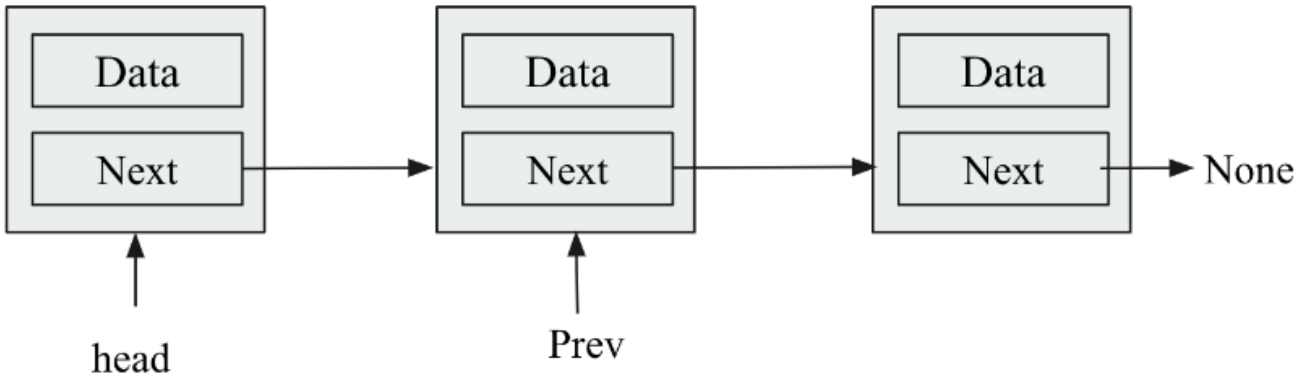


CMC UNIVERSITY

Linked List Operations - Delete

Deleting any intermediate node in a singly linked list

- Finally, the list after deleting the node



Let’s say we want to delete a data element that has the given value. For this given condition, we can first search the node to be deleted and then delete the node as per the steps discussed.

CMC UNIVERSITY

Linked List Operations - Delete

```
def delete(self, data):
    current = self.head
    prev = self.head
    while current:
        if current.data == data:
            if current == self.head:
                self.head = current.next
            else:
                prev.next = current.next
            self.size -= 1
            return
        prev = current
        current = current.next
```

Assuming that we already have a linked list of three items – *eggs*, *ham*, *spam*, the following code is for deleting a data element with the value "*ham*" from the given linked list:

```
words.delete("ham")
current = words.head
while current:
    print(current.data)
    current = current.next
```

Output:

eggs
spam

The worst-case time complexity of the delete operation is O(n)

CMC UNIVERSITY

Linked List Operations - Delete

Clearing a list

We may need to clear a list quickly, and there is a very simple way to do this. We can clear a list by simply clearing the pointer *head* and *tail* by setting them to *None*

```
def clear(self):
    # clear the entire list.
    self.tail = None
    self.head = None
```

CMC UNIVERSITY

Linked List Applications

- Dynamic memory allocation
- Implemented in stack and queue
- In undo functionality of softwares
- Hash tables, Graphs

CMC UNIVERSITY

STACKS

CMC UNIVERSITY

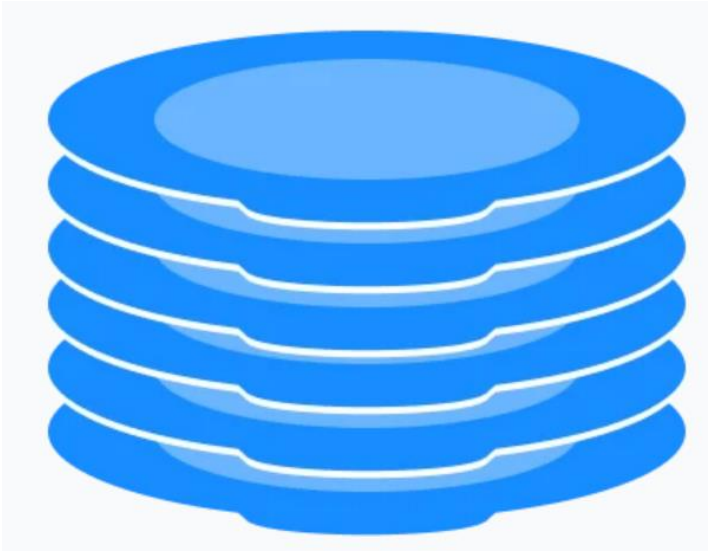
Stacks

- A stack is a linear data structure that follows the principle of **Last In First Out (LIFO)**. This means the last element inserted inside the stack is removed first.
- You can think of the stack data structure as the pile of plates on top of another.

A stack is a data structure that stores the data in a specific order similar with several constraints:

- Data elements in a stack can only be inserted at the end (push operation)
- Data elements in a stack can only be deleted from the end (pop operation)
- Only the last data element can be read from the stack (peek operation)

If you want the plate at the bottom, you must first remove all the plates on top.



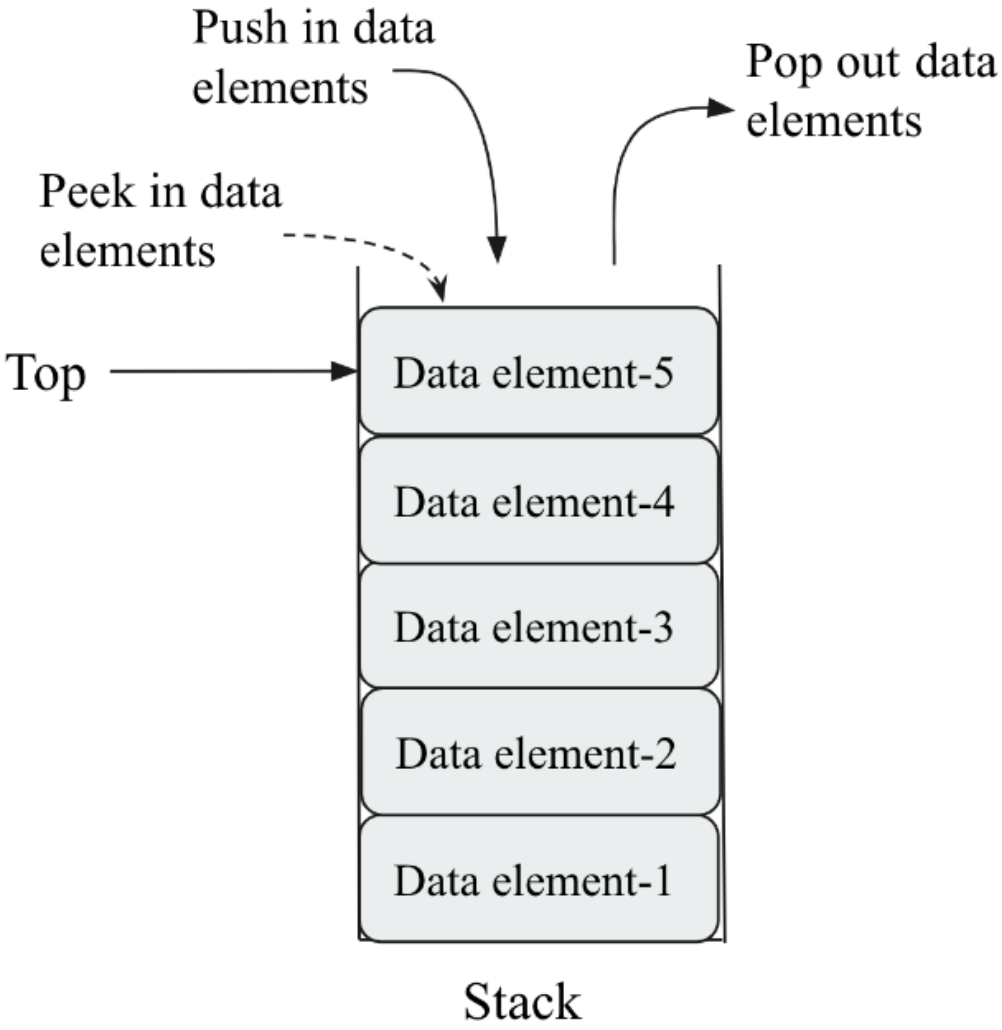
Stack representation similar to a pile of plate

CMC UNIVERSITY

Stacks

LIFO Principle of Stack

- In programming terms, putting an item on top of the stack is called **push** and removing an item is called **pop**.
- All the operations in the stack are performed through a pointer, which is generally named *Top*



Stack Push and Pop Operations

CMC UNIVERSITY

Stacks

The following table demonstrates the use of two important stack operations (push and pop) in the stack:

Stack operation	Size	Contents	Operation results
`stack()`	0	`[]`	Stack object created, which is empty.
`push "egg"`	1	`['egg']`	One item egg is added to the stack.
`push "ham"`	2	`['egg', 'ham']`	One more item, ham, is added to the stack.
`peek()`	2	`['egg', 'ham']`	The top element, ham, is returned.
`pop()`	1	`['egg']`	The ham item is popped off and returned. (This item was added last, so it is removed first.)
`pop()`	0	`[]`	The egg item is popped off and returned. (This is the first item added, so it is returned last.)

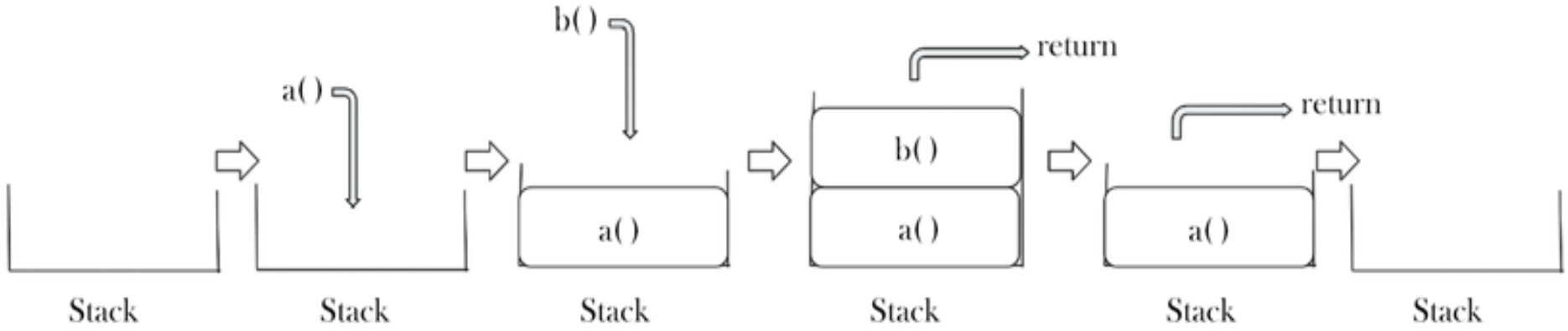
CMC UNIVERSITY

Stacks

Stacks are used for a number of things. One common usage for stacks is to keep track of the return address during function calls. Let's imagine that we have the following program:

```
def b():
    print('b')

def a():
    b()

a()
print("done")
```

When the program execution gets to the call to a(), a sequence of events will be followed in order to complete the execution of this program. A visualization of all these steps is shown in Figure:



Stacks

The sequence of events is as follows:

1. First, the address of the current instruction is pushed onto the stack, and then execution jumps to the definition of *a*
2. Inside function *a()*, function *b()* is called
3. The return address of function *b()* is pushed onto the stack. Once the execution of the instructions and functions in *b()* are complete, the return address is popped off the stack, which takes us back to function *a()*
4. When all the instructions in function *a()* are completed, the return address is again popped off the stack, which takes us back to the main program and the print statement

The output:

b

done

Stacks

Basic Operations of Stack

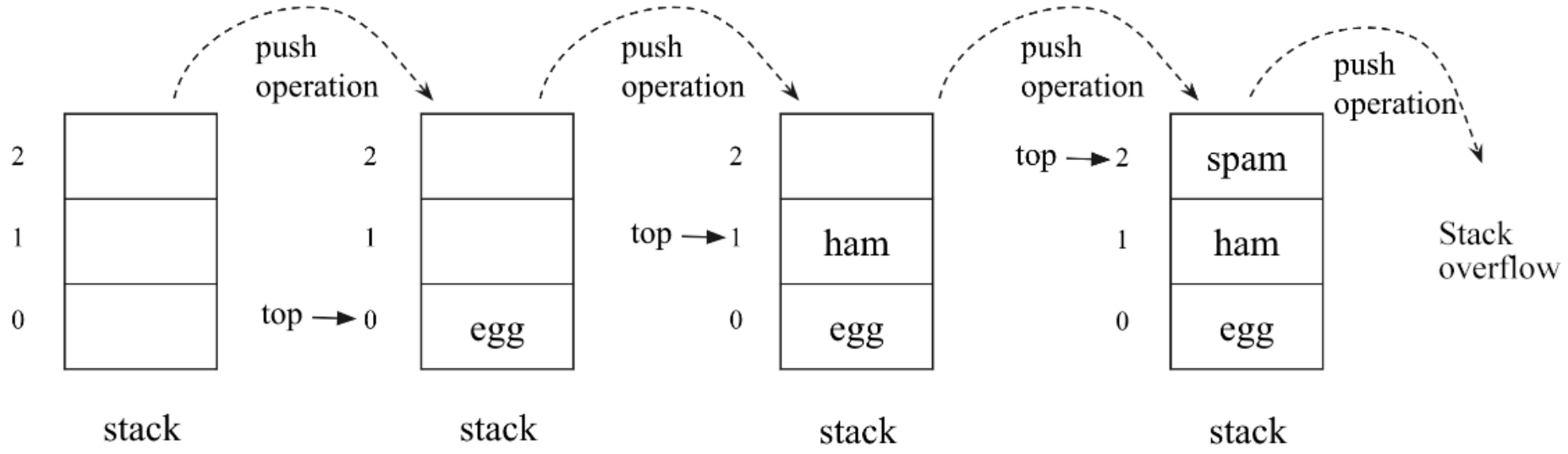
There are some basic operations that allow us to perform different actions on a stack.

- **Push**: Add an element to the top of a stack
- **Pop**: Remove an element from the top of a stack
- **IsEmpty**: Check if the stack is empty
- **IsFull**: Check if the stack is full
- **Peek**: Get the value of the top element without removing it

Stack implementation using arrays

In the case of the array-based implementation of a stack (where the stack has a fixed size), it is important to check whether the stack is full or not, since trying to push an element into a full stack will generate an error, called an overflow. Likewise, trying to apply a pop operation to an empty stack causes an error known as an underflow.

- An overview of the push operation on a stack using an array is shown in the Figure



CMC UNIVERSITY

Stack implementation using arrays

```
size = 3
data = [0]*(size)    #Initialize the stack
```

```
top = -1

def push(x):
    global top
    if top >= size - 1:
        print("Stack Overflow")
    else:
        top = top + 1
        data[top] = x
```

```
push('egg')
push('ham')
push('spam')

print(data[0 : top + 1] )

push('new')
push('new2')
```

The output:

['egg', 'ham', 'spam']
Stack Overflow
Stack Overflow

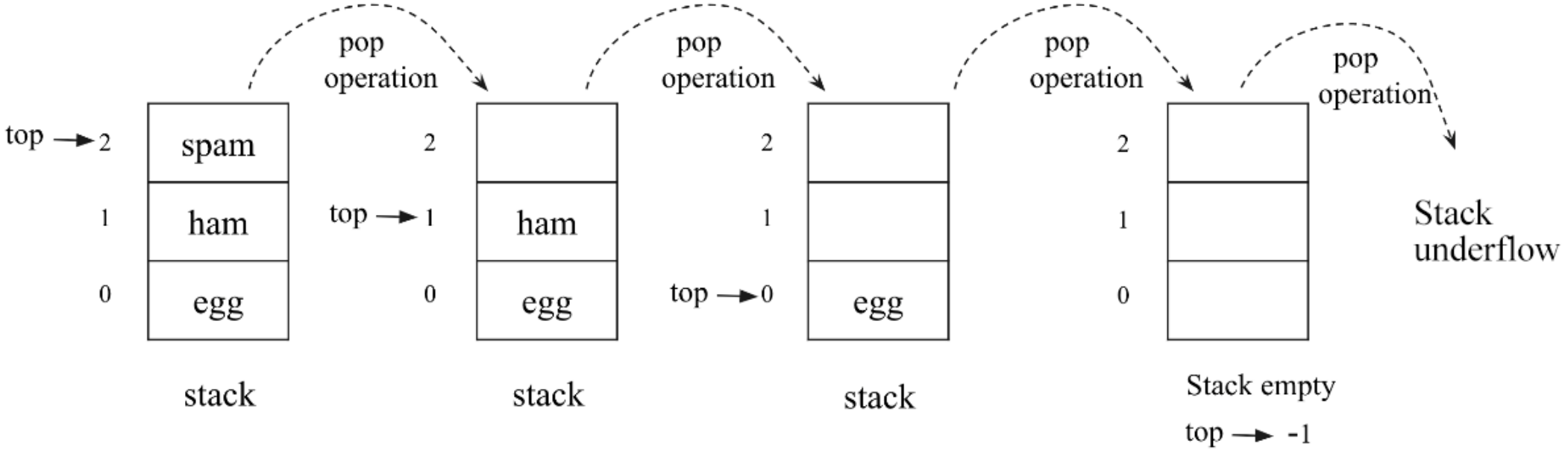
CMC UNIVERSITY

Stack implementation using arrays

We initialize the stack with a fixed size (size = 3), the top pointer to –1, which indicates that the stack is empty.
In the push method, the top pointer is compared with the size of the stack to check the overflow condition:
- If the stack is full, the stack overflow message is printed.
- If the stack is not full, the top pointer is incremented by 1, and the new data element is added to the top of the stack.

Stack implementation using arrays

- An overview of the pop operation on a stack using an array is shown in the Figure



CMC UNIVERSITY

Stack implementation using arrays

```
def pop():
    global top
    if top == -1:
        print("Stack Underflow")
    else:
        top = top - 1
        data[top] = 0
        return data[top+1]
```

We first check the underflow condition by checking whether the stack is empty or not:

- If the top pointer has a value of –1, it means the stack is empty.
- Otherwise, the data elements in the stack are removed by decrementing the top pointer by 1, and the top data element is returned to the main function

```
print(data[0 : top + 1])
pop()
pop()
pop()
pop()
print(data[0 : top + 1])
```

Let's assume we already added three data elements to the stack, and then we call the pop function four times. Since there are only three elements in the stack, the initial three data elements are removed, and when we try to call the pop operation a fourth time, the stack underflow message is printed.

CMC UNIVERSITY

Stack implementation using arrays

- An overview of the peek operation on a stack using an array is shown in the Python code:

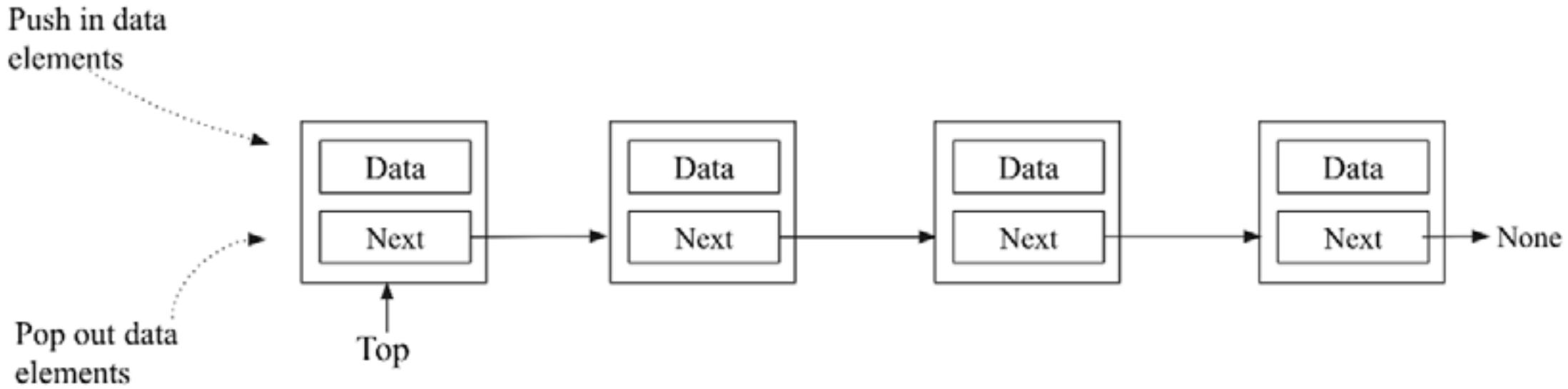
```
def peek():
    global top
    if top == -1:
        print("Stack is empty")
    else:
        print(data[top])
```

We check the position of the top pointer in the stack:

- If the value of the top pointer is –1, it means that the stack is empty,
- Otherwise, we print the value of the top element of the stack.

Stack implementation using linked lists

Implementing the stack data structure using a linked list can be treated as a standard linked list with some constraints, including that elements can be added or removed from the end of the list (push and pop operations) through the top pointer



Stack using linked list

CMC UNIVERSITY

Stack implementation using linked lists

We need two things to implement a stack:

1. We first need to know which node is at the top of the stack so that we can apply the push and pop operations through this node

2. We would also like to keep track of the number of nodes in the stack, so we add a size variable to the Stack class

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

```
class Stack:
    def __init__(self):
        self.top = None
        self.size = 0
```

CMC UNIVERSITY

Stack implementation using linked lists

Push operation

We check if the stack already has some items in it or if it is empty. We are not required here to check the overflow condition because we are not required to fix the length of the stack, unlike the stack implementation using arrays.

If the stack already has some elements, then we have to do two things:

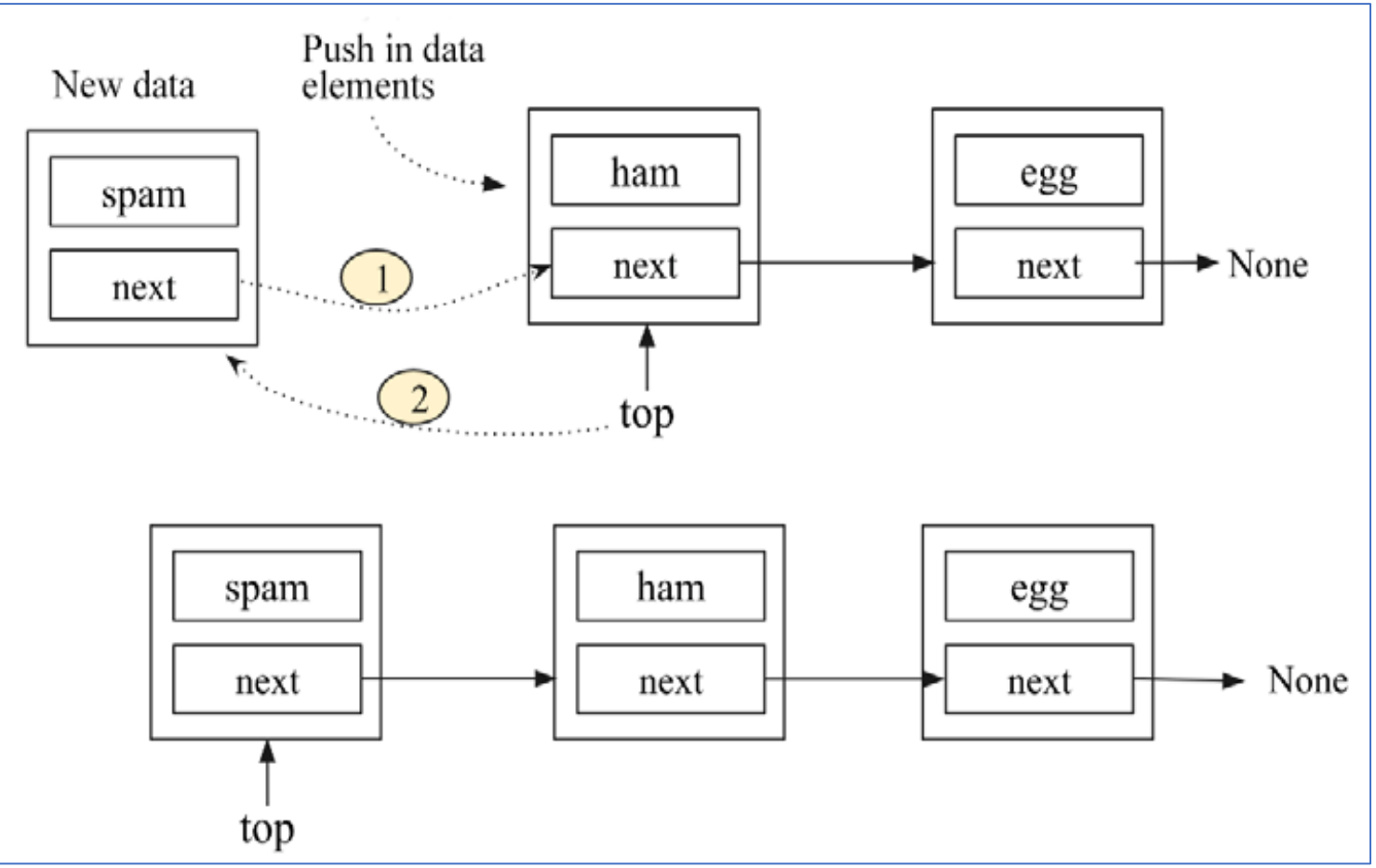
1. The new node must have its next pointer pointing to the node that was at the top earlier

2. We put this new node at the top of the stack by pointing self.top to the newly added node

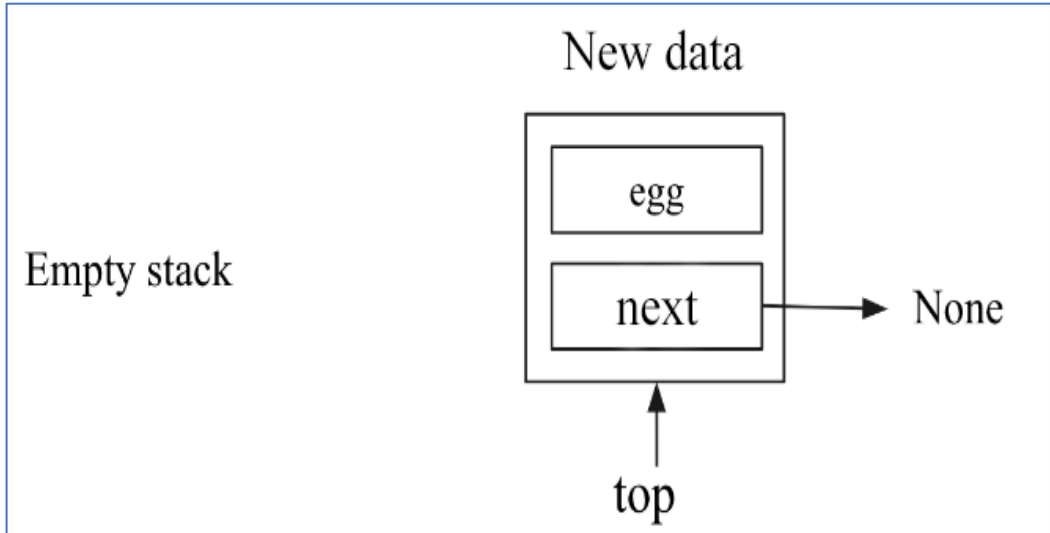
CMC UNIVERSITY

Stack implementation using linked lists

Workings of the push operation on the stack



Insertion of the data element “egg” into an empty stack



CMC UNIVERSITY

Stack implementation using linked lists

```
def push(self, data):
    # create a new node
    node = Node(data)
    if self.top:
        node.next = self.top
        self.top = node
    else:
        self.top = node
    self.size += 1
```

We create a new node and store the data in that. Then we check the position of the top pointer:

- If it is not null, that means the stack is not empty, and we add the new node, updating two pointers.
- Otherwise, we make the top pointer point to the new node.

Finally, we increase the size of the stack by incrementing the *self.size* variable.

CMC UNIVERSITY

Stack implementation using linked lists

```
words = Stack()
words.push('egg')
words.push('ham')
words.push('spam')

#print the stack elements.
current = words.top
while current:
      print(current.data)
      current = current.next
```

We create a stack of three data elements

Output:

spam
ham
egg

Stack implementation using linked lists

Pop operation

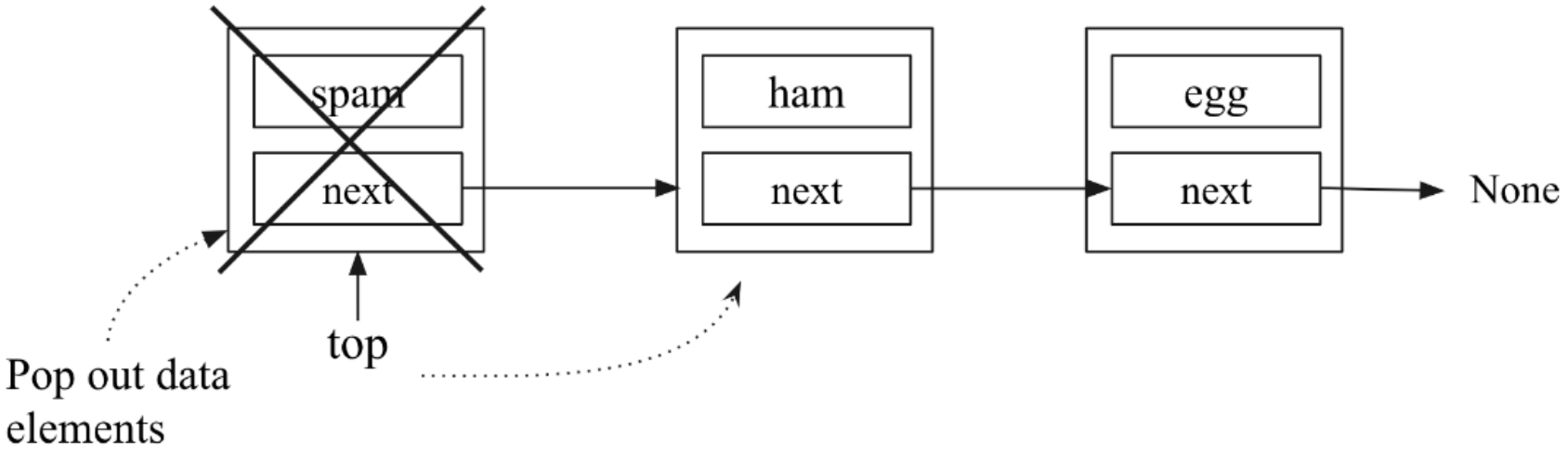
In this operation, the topmost element of the stack is read, and then removed from the stack. The pop method returns the topmost element of the stack and returns None if the stack is empty.

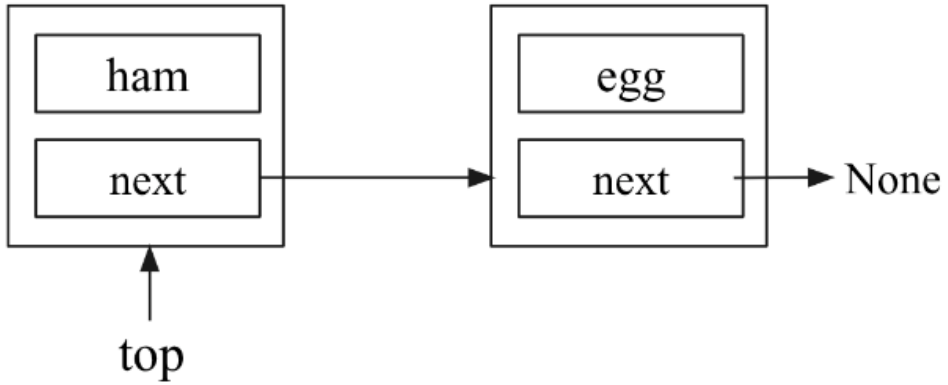
To implement the pop operation on a stack, we do following:

1. First, check if the stack is empty. The pop operation is not allowed on an empty stack.
2. If the stack is not empty, check whether the top node has its *next* attribute pointing to some other node. If so, it means the stack contains elements, and the topmost node is pointing to the next node in the stack. To apply the pop operation, we have to change the top pointer. The next node should be at the top. We do this by pointing *self.top* to *self.top.next*.

Stack implementation using linked lists

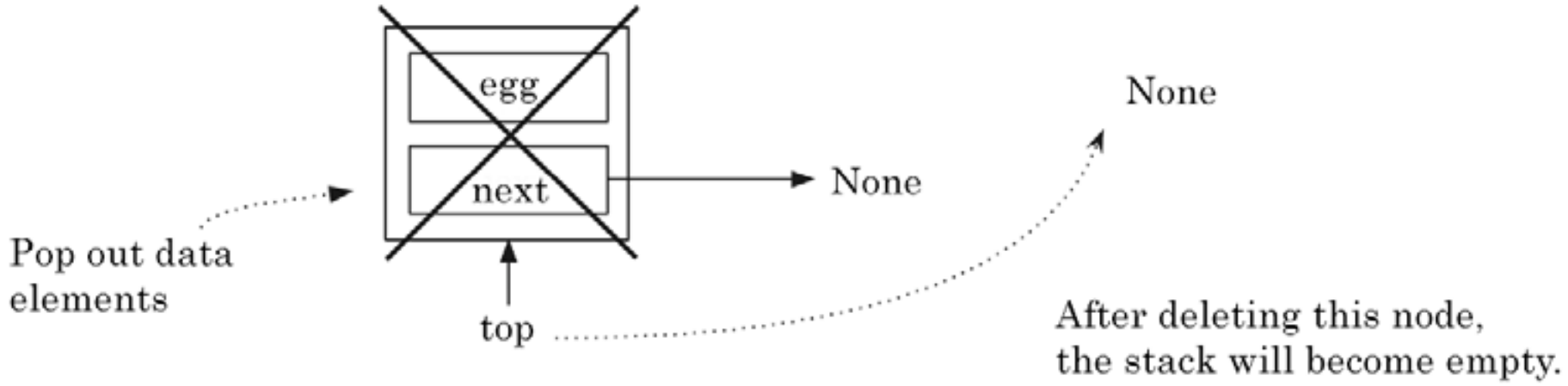
Workings of the pop operation on the stack





Stack implementation using linked lists

3. When there is only one node in the stack, the stack will be empty after the pop operation. We have to change the top pointer to None



4. Removing this node results in self.top pointing to None
5. We also decrement the size of the stack by 1 if the stack is not empty.

CMC UNIVERSITY

Stack implementation using linked lists

```
def pop(self):
    if self.top:
        data = self.top.data
        self.size -= 1
        if self.top.next:  #check if there is more than one node.
            self.top = self.top.next
        else:
            self.top = None
        return data
    else:
        print("Stack is empty")
```

We check the position of the top pointer: If it is not null (the stack is not empty):
- if there is more than one data element in the stack, we move the top pointer to point to the next node
 - if that is the last node, we make the top pointer point to None

We also decrease the size of the stack by decrementing the self.size variable.

CMC UNIVERSITY

Stack implementation using linked lists

Let's say we have three data elements in a stack. We can use the following code to apply the pop operation to the stack:

```
words.pop()
current = words.top
while current:
    print(current.data)
    current = current.next
```

Output:

ham
egg

In the above code, we popped off the top element from the stack of three elements – egg, ham, spam.

Stack implementation using linked lists

Peek operation

This method returns the top element from the stack without deleting it from the stack.

The only difference between peek and pop is that the peek method just returns the topmost element; however, in the case of a pop method, the topmost element is returned, and that element is also deleted from the stack

```
def peek(self):
    if self.top:
        return self.top.data
    else:
        print("Stack is empty")
```

We first check the position of the top pointer using self.top:
- If it is not null (the stack is not empty), return the data value of the topmost node,
- Otherwise, print the message that the stack is empty.

We can use the peek method to get the top element of the stack through the following code

```
words.peek()
```

Output:

spam

Applications of stacks

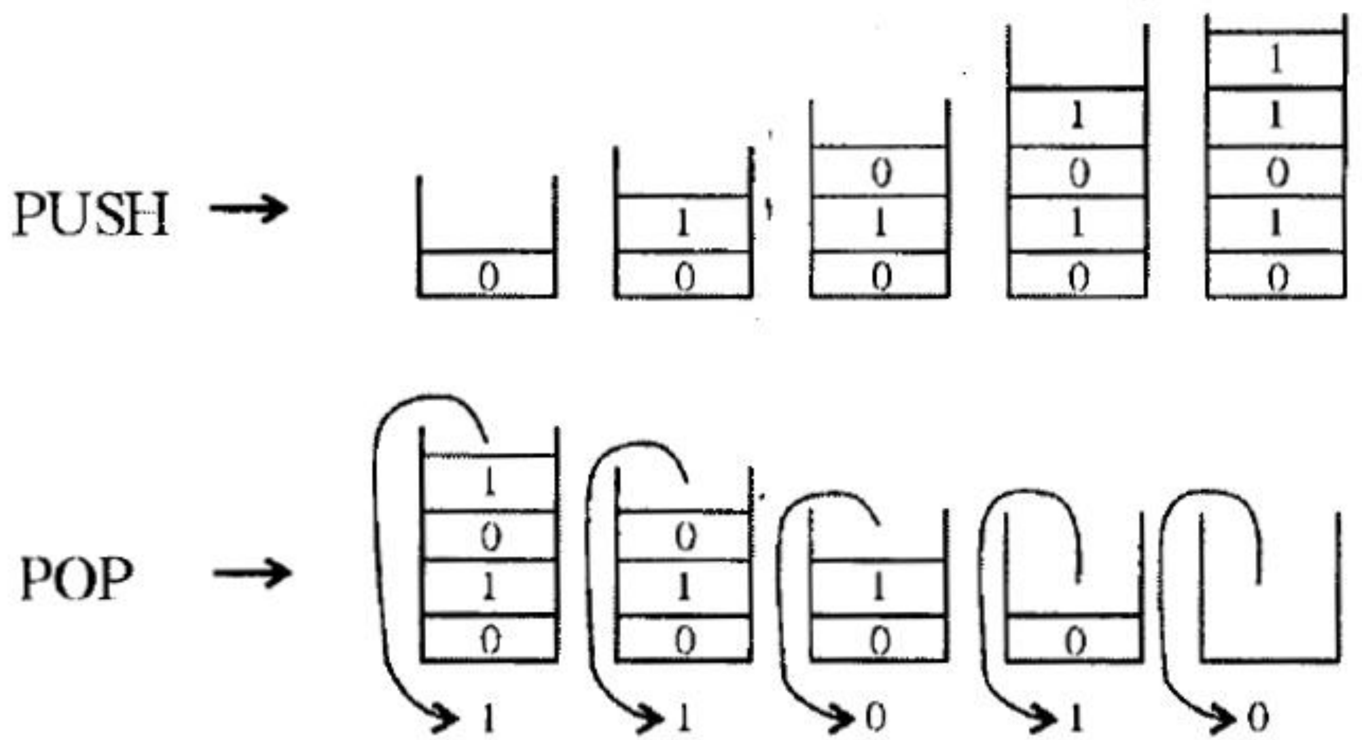
Stack Time Complexity

For the array-based implementation of a stack, the push and pop operations take constant time, i.e. O(1).

- ***Application in radix conversion***

Example:

$(26)_{10} \rightarrow (01011) \rightarrow (11010)_2$



CMC UNIVERSITY

Applications of stacks

- ***Bracket-matching of a given expression***

Write a function check_brackets that will verify whether a given expression containing brackets—( ), [ ], or { }— is balanced or not, that is, whether the number of closing brackets matches the number of opening brackets

The above function parses each character in the expression passed to it:

- If it gets an open bracket, it pushes it onto the stack.
- If it gets a closing bracket, it pops the top element off the stack and compares the two brackets to make sure their types match (should match), [should match], and { should match }. If they don't, we return False; otherwise, we continue parsing.

Once we reach the end of the expression, we need to do one last check. If the stack is empty, then it is fine and we can return True. But if the stack is not empty, then we have an opening bracket that does not have a matching closing bracket and we will return False.

CMC UNIVERSITY

Applications of stacks

- ***Arithmetic expressions and Polish notation***

The way to write arithmetic expression is known as a notation. An arithmetic expression can be written in three different but equivalent notations, i.e., without changing the essence or output of an expression. These notations are –*Infix*, *Prefix*, *Postfix*

Expression No	Infix Notation	Prefix Notation	Postfix Notation
1	a + b	+ a b	a b +
2	(a + b) * c	* + a b c	a b + c *
3	a * (b + c)	* a + b c	a b c + *
4	a / b + c / d	+ / a b / c d	a b / c d / +
5	(a + b) * (c + d)	* + a b + c d	a b + c d + *
6	((a + b) * c) - d	- * + a b c d	a b + c * d -

CMC UNIVERSITY

Applications of stacks

- ***Arithmetic expressions and Polish notation***

Precedence and associativity determines the order of evaluation of an expression:
^, *, /, +, -

We shall now look at the algorithm on how to evaluate postfix notation:

Step 1 – scan the expression from left to right

Step 2 – if it is an operand push it to stack

Step 3 – if it is an operator pull operand from stack and perform operation

Step 4 – store the output of step 3, back to stack

Step 5 – scan the expression until all operands are consumed

Step 6 – pop the stack and perform operation

CMC UNIVERSITY

Applications of stacks

- ***Arithmetic expressions and Polish notation***

```
def postfix_evaluation(expression):
    stack = []

    # Helper function to perform arithmetic operations
    def perform_operation(operator, operand2, operand1):
        if operator == '+':
            return operand1 + operand2
        elif operator == '-':
            return operand1 - operand2
        elif operator == '*':
            return operand1 * operand2
        elif operator == '/':
            return operand1 / operand2
```

```
    # Iterate through each character in the expression
    for char in expression:
        if char.isdigit():
            stack.append(int(char))
        else:
            operand2 = stack.pop()
            operand1 = stack.pop()
            result = perform_operation(char, operand2, operand1)
            stack.append(result)
```

```
    # The final result will be at the top of the stack
    return stack.pop()
```

```
# Example usage
postfix_expression = "34+2*"
result = postfix_evaluation(postfix_expression)
print("Result:", result)
```

CMC UNIVERSITY

Applications of stacks

- **To reverse a word -** Put all the letters in a stack and pop them out. Because of the LIFO order of stack, you will get the letters in reverse order.
- **In compilers -** Compilers use the stack to calculate the value of expressions like 2 + 4 / 5 * (7 - 9) by converting the expression to prefix or postfix form.
- **In browsers -** The back button in a browser saves all the URLs you have visited previously in a stack. Each time you visit a new page, it is added on top of the stack. When you press the back button, the current URL is removed from the stack, and the previous URL is accessed.

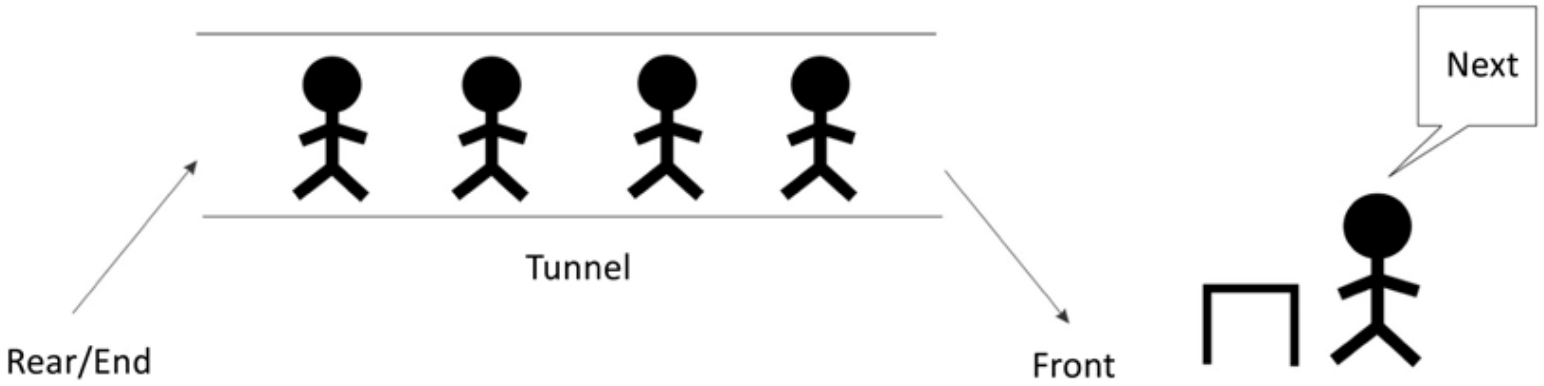
CMC UNIVERSITY

QUEUES

CMC UNIVERSITY

Queues

- A queue is a useful data structure in programming. It is similar to the ticket queue outside a cinema hall, where the first person entering the queue is the first person who gets the ticket.
- Queue follows the **First In First Out (FIFO)** rule - the item that goes in first is the item that comes out first.



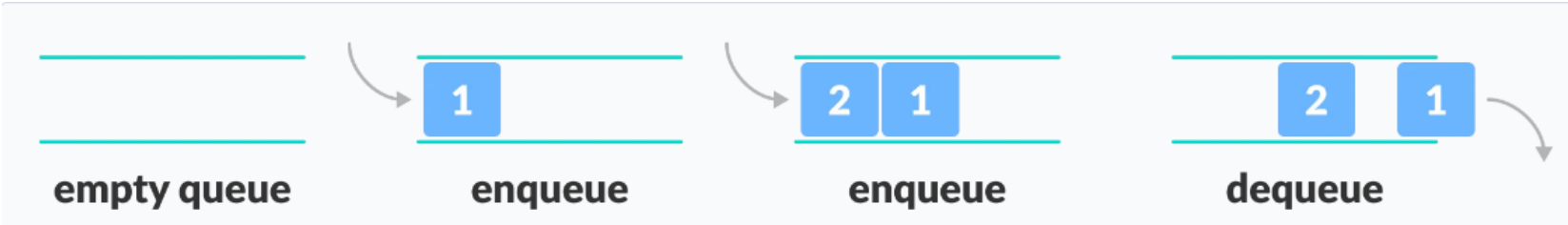
To join the queue, participants must stand behind the last person in the queue. This is the only legal or permitted way the queue accepts new entrants. The length of the queue does not matter.

CMC UNIVERSITY

Queues

A queue is a list of elements stored in sequence with the following constraints:

- Data elements can only be inserted from one end, the rear end/tail of the queue.
- Data elements can only be deleted from the other end, the front/head of the queue.
- Only data elements from the front of the queue can be read.



FIFO Representation of Queue

- In the above image, since 1 was kept in the queue before 2, it is the first to be removed from the queue as well. It follows the **FIFO** rule.
- In programming terms, putting items in the queue is called **enqueue**, and removing items from the queue is called **dequeue**. Whenever an element is enqueued, the length or size of the queue increments by 1, and dequeuing an item reduces the number of elements in the queue by 1.

Queues

Basic Operations of Queue

A queue is an object (an abstract data structure - ADT) that allows the following operations:

- **Enqueue**: Add an element to the end of the queue
- **Dequeue**: Remove an element from the front of the queue
- **IsEmpty**: Check if the queue is empty
- **IsFull**: Check if the queue is full
- **Peek**: Get the value of the front of the queue without removing it

CMC UNIVERSITY

Python's list-based queues

- Firstly, in order to implement a queue based on Python's list data structure, we create a ListQueue class, in which we declare and define the different functionalities of queue. In this method, we store the actual data in Python's list data structure. The ListQueue class is defined as follows:

```
class ListQueue:
    def __init__(self):
        self.items = []
        self.front = self.rear = 0
        self.size = 3      # maximum capacity of the queue
```

- In the __init__ initialization method, the items instance variable is set to [] (the queue is empty when created).
- The size of the queue is also set to 4 (as an example in this code), which is the maximum capacity for the number of elements that can be stored in the queue.
- The initial position of the rear and front indices are also set to 0.

Python's list-based queues

Working of Queue

Queue operations work as follows:

- two pointers FRONT and REAR
- FRONT track the first element of the queue
- REAR track the last element of the queue
- initially, set value of FRONT and REAR to -1

CMC UNIVERSITY

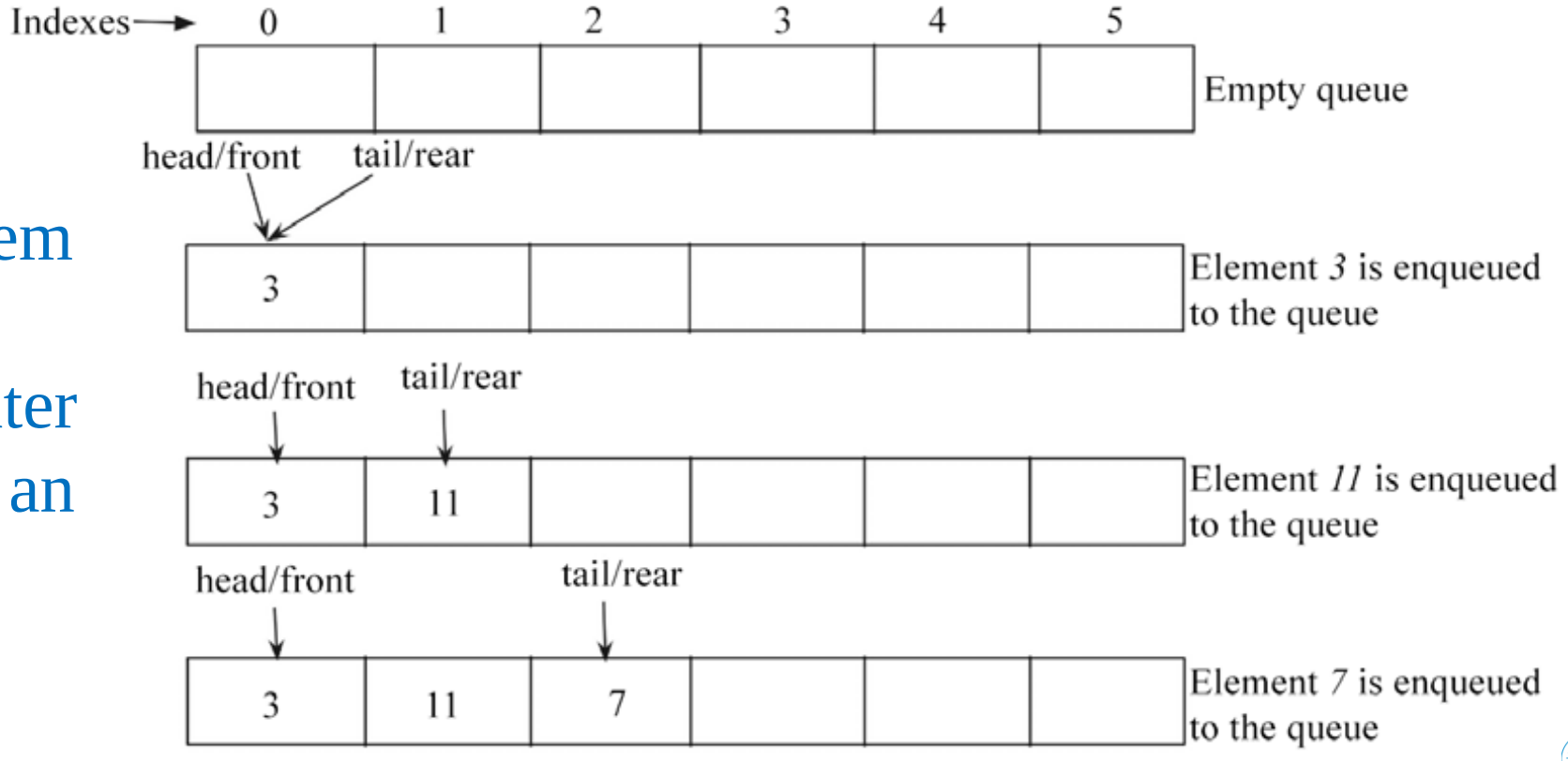
Enqueue Operation

- check if the queue is full
- for the first element, set the value of FRONT to 0
- increase the REAR index by 1
- add the new element in the position pointed to by REAR

Enqueue Operation

- The enqueue operation adds an item at the end of the queue. Consider the example of adding elements to the queue to understand the concept shown in the following Figure. We start with an empty list. Initially, we add an item 3 at index 0.

Next, we add an item 11 at index 1, and move the rear pointer every time we add an element:



CMC UNIVERSITY

Enqueue Operation

- In order to implement the enqueue operation, we use the append method of the List class to append items (or data) to the end of the queue. See the following code for the implementation of the enqueue method. This should be defined in the ListQueue class:

We first check whether the queue is full by comparing the maximum capacity of the queue with the position of the rear index. Further, if there is space in the queue, we use the append method of the List class to add the data at the end of the queue and increase the rear pointer by 1.

```
def enqueue(self, data):
    if self.size == self.rear:
        print("\n Queue is full")
    else:
        self.items.append(data)
        self.rear += 1
```

Enqueue Operation

To create a queue using the ListQueue class, we use the following code:

```
q= ListQueue()
q.enqueue(20)
q.enqueue(30)
q.enqueue(40)
q.enqueue(50)


print(q.items)
```

In this code, we add can a maximum of three data elements since we have set the maximum capacity of the queue to be 3. After adding three elements, when we try to add another new element, we get a message that the queue is full.

The output:

Queue is full
[20, 30, 40]

Dequeue Operation

Dequeue Operation

- check if the queue is empty
- return the value pointed by FRONT
- increase the FRONT index by 1
- for the last element, reset the values of FRONT and REAR to -1

CMC UNIVERSITY

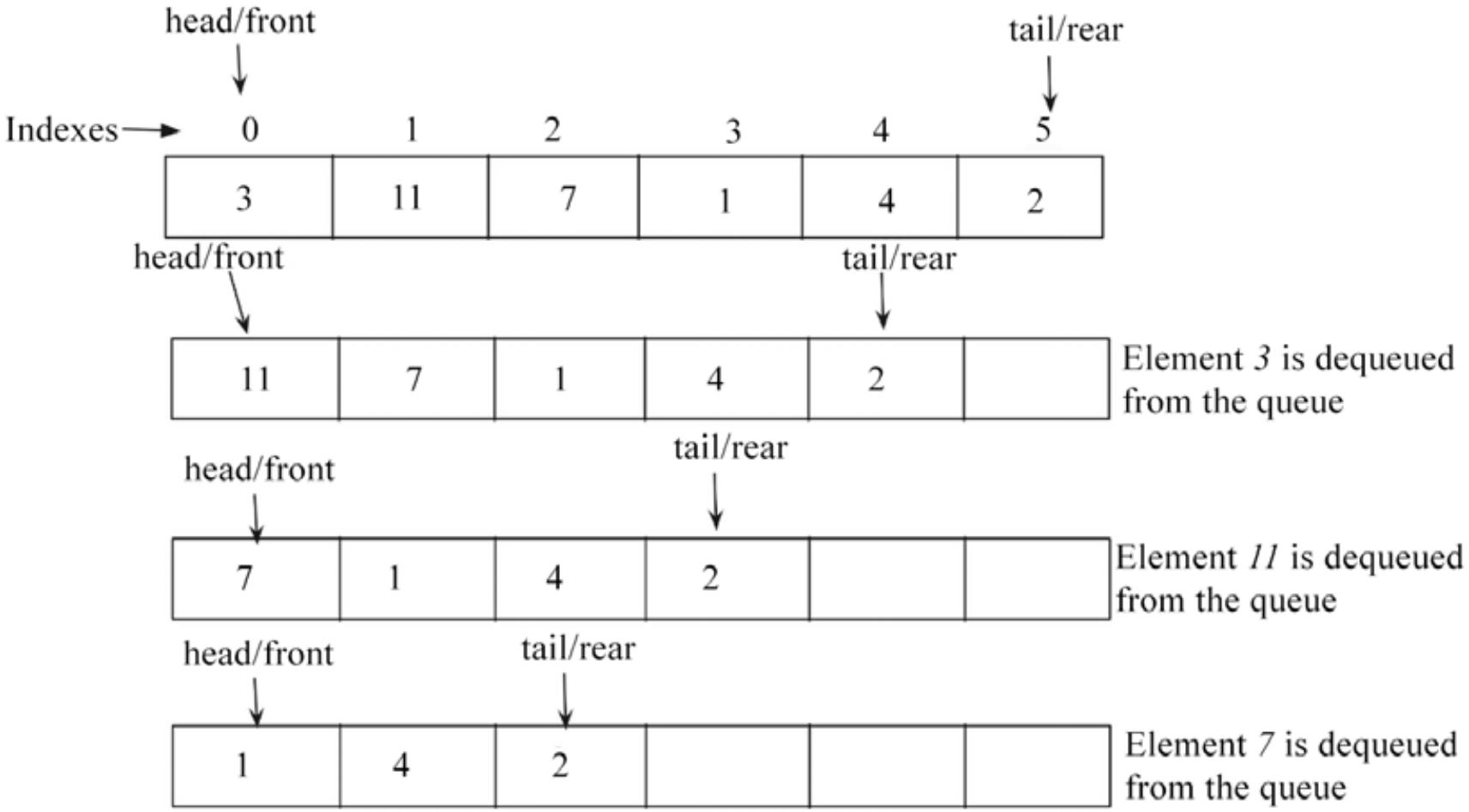
Dequeue Operation

- The dequeue operation is used to read and delete items from the queue. This method returns the front item from the queue and deletes it. Consider the example of dequeuing elements from the queue shown in Figure 5.14. Here, we have a queue containing elements {3, 11, 7, 1, 4, 2}.
- In order to dequeue any element from this queue, the element inserted first will be removed first, so the item 3 is removed. When we dequeue any element from the queue, we also decrease the rear pointer by 1:

CMC UNIVERSITY

Dequeue Operation

Example of a dequeue operation on a queue



CMC UNIVERSITY

Dequeue Operation

The following is the implementation of the dequeue method, which should be defined in the ListQueue class:

```
    def dequeue(self):
        if self.front == self.rear:
            print("Queue is empty")
        else:
            data = self.items.pop(0)    # delete the item from front end
of the queue
            self.rear -= 1
            return data
```

In this code, we firstly check whether the queue is already empty by comparing the front and rear pointers:

- If both rear and front pointers are same, it means the queue is empty.
- If there are some elements in the queue, we use the pop method to dequeue an element.

CMC UNIVERSITY

Dequeue Operation

The Python List class has a method called pop(). The pop method does the following:

1. Deletes the last item from the list
2. Returns the deleted item from the list back to the user or code that called it

The item at the first position pointed to by the front variable is popped and saved in the data variable. We also decrease the rear variable by 1, since one data item has been deleted from the queue. Finally, in the last line of the method, the data is returned.

```
data = q.dequeue()
print(data)
print(q.items)
```

The output:

```
20
[30, 40]
```

CMC UNIVERSITY

Linked list based queues

A queue data structure can also be implemented using any linked list, such as singly-linked or doubly-linked lists. We implement queues using linked lists that follow the FIFO property of the queue data structure.
The Queue class is very similar to that of the doubly-linked list class. Here, we have head and tail pointers, where tail points to the end of the queue (the rear end) that will be used for adding new elements, and the head pointer points to the start of the queue (the front end) that will be used for dequeuing the elements from the queue.

CMC UNIVERSITY

Linked list based queues

Initially, the self.head and self.tail pointers are set to None upon creation of an instance of the Queue class. To keep a count of the number of nodes in Queue, the count instance variable is also maintained here and initially set to 0.

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

class Queue:
    def __init__(self):
        self.head = None
        self.tail = None
        self.count = 0
```

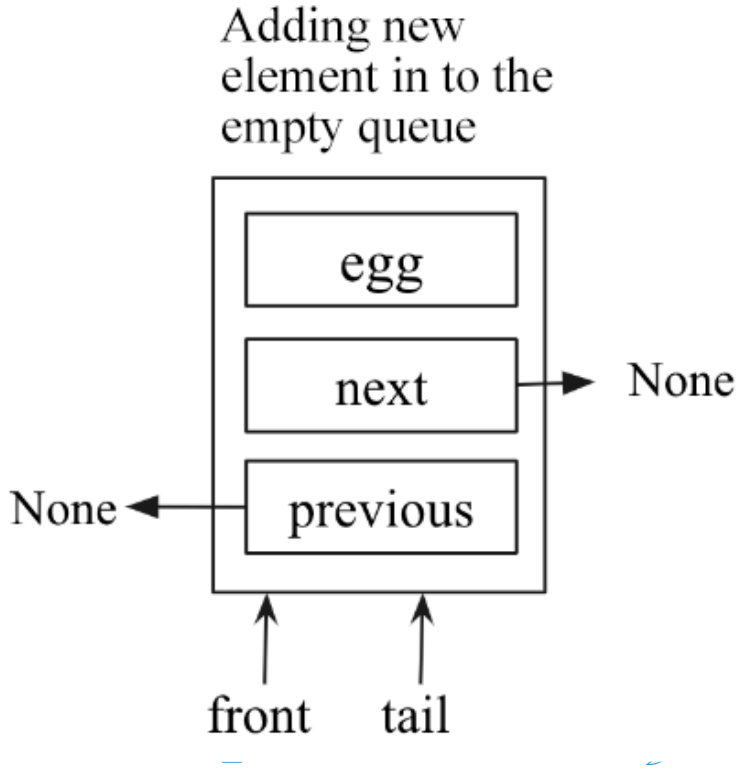
CMC UNIVERSITY

Linked list based queues

The enqueue operation

Elements are added to a Queue object via the enqueue method. The data elements are added through nodes.

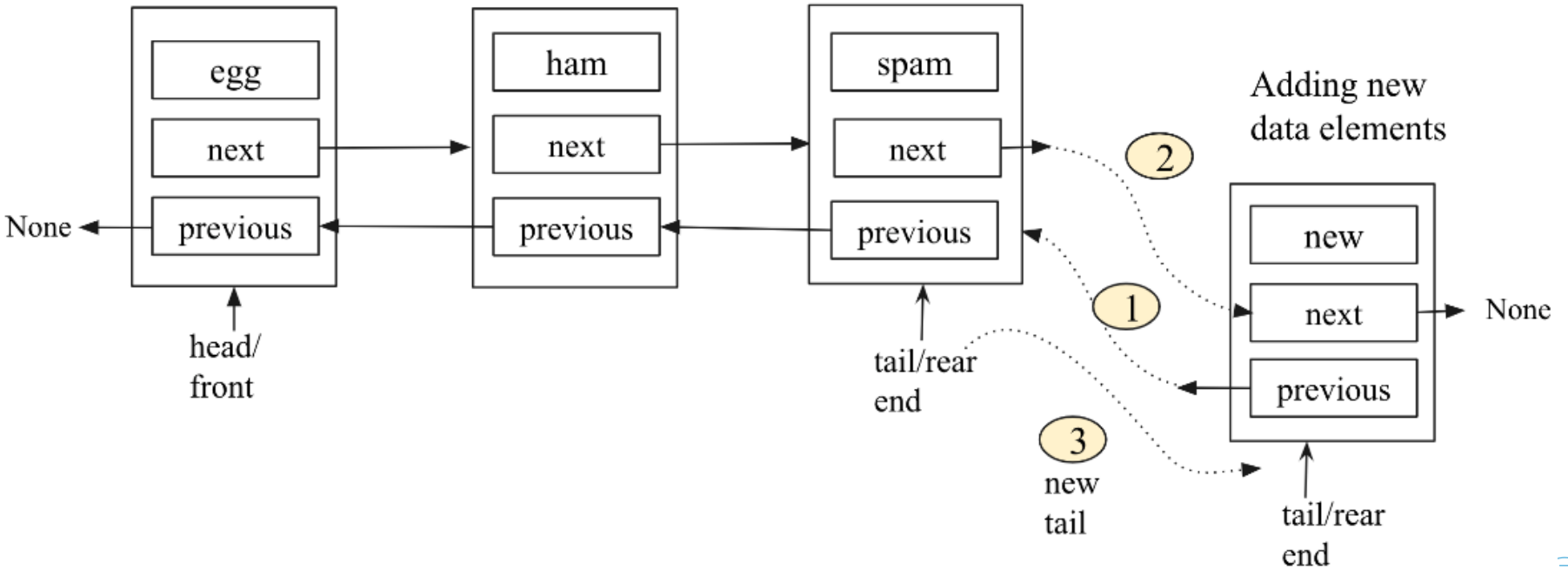
- Create a new node
- If the queue is empty, the new node becomes the first node of the queue, as shown in the Figure
- Otherwise, the new node is appended to the rear end of the queue.



Linked list based queues

The enqueue operation

We append the node by updating three links: (1) the previous pointer of the new node should point to the tail of the queue, (2) the next pointer of the tail node should point to the new node, and (3) the tail pointer should be updated to the new node.



CMC UNIVERSITY

Linked list based queues

```
def enqueue(self, data):
    new_node = Node(data, None, None)
    if self.head == None:
        self.head = new_node
        self.tail = self.head
    else:
        new_node.prev = self.tail
        self.tail.next = new_node
        self.tail = new_node

    self.count += 1
```

- We first check whether the queue is empty or not:
  + If head points to None: the new node is made the first node of the queue, and we make both self.head and self.tail point to the newly created node.
  + Otherwise, append the new node to the rear of the queue by updating the three links
- Finally, the total count of elements in the queue is increased by 1.

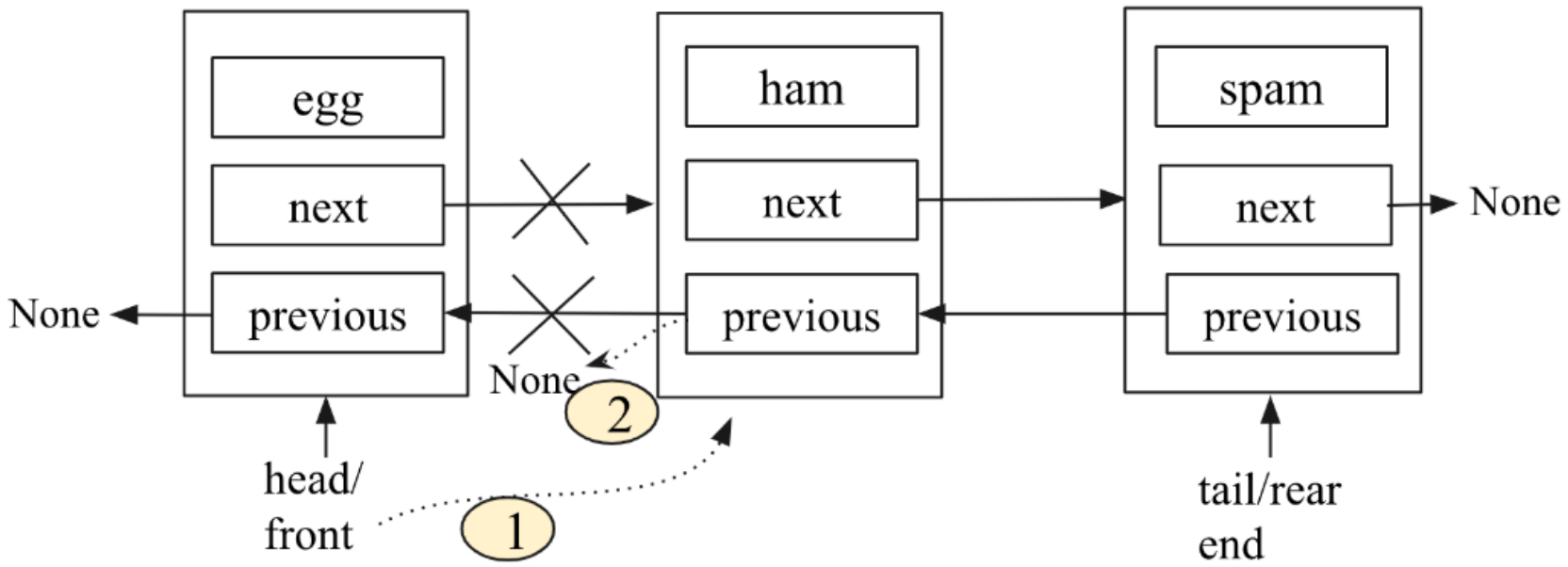
The worst-case time complexity of an enqueue operation on the queue is O(1), since any item can be appended directly through the tail pointer in constant time

Linked list based queues

The dequeue operation

This method removes the node at the front of the queue.

- We first check whether the dequeuing element is the last node in the queue:
  + If it is the last node, make the queue empty after the dequeue operation
  + Otherwise, dequeue the first element by updating the front/head pointer to the next node and the previous pointer of the new head to None:



CMC UNIVERSITY

Linked list based queues

The dequeue operation

- We firstly check the number of items in the queue using the self.count variable.
  + If self.count = 1 (the dequeuing element is the last element), update the head and tail pointers to None.
  + If the queue has many nodes, then the head pointer is shifted to point to the next node after self.head by updating the two links. We also check whether there is an item left in the queue, and if not, then a message is printed that the queue is empty.
- Finally, the self.count variable is decremented by 1.

```
def dequeue(self):
    if self.count == 1:
        self.count -= 1
        self.head = None
        self.tail = None
    elif self.count > 1:
        self.head = self.head.next
        self.head.prev = None
    elif self.count <1:
        print("Queue is empty")
    self.count -= 1
```

The worst-case time complexity of a dequeue operation in the queue is O(1).

CMC UNIVERSITY

Queues

Complexity Analysis

- The complexity of enqueue and dequeue operations in a queue using an array is O(1). If you use pop(N) in python code, then the complexity might be O(n) depending on the position of the item to be popped.

Applications of Queue

- CPU scheduling, Disk Scheduling
- When data is transferred asynchronously between two processes.The queue is used for synchronization. For example: IO Buffers, pipes, file IO, etc
- Handling of interrupts in real-time systems.
- Call Center phone systems use Queues to hold people calling them in order.

CMC UNIVERSITY

Queues

Types of Queues

A queue is a useful data structure in programming. It is similar to the ticket queue outside a cinema hall, where the first person entering the queue is the first person who gets the ticket.

There are four different types of queues:

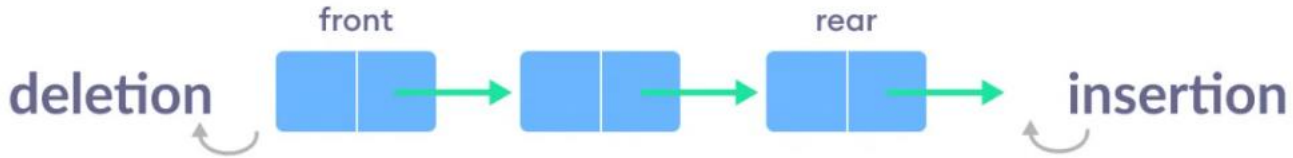
- Simple Queue
- Circular Queue
- Priority Queue
- Double Ended Queue

CMC UNIVERSITY

Queues

Simple Queue

- In a simple queue, insertion takes place at the rear and removal occurs at the front. It strictly follows the FIFO (First in First out) rule.

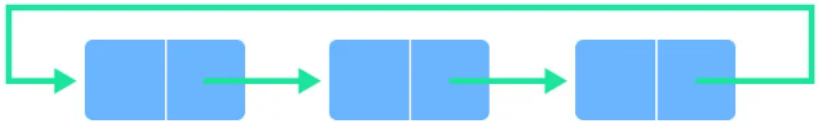


Simple Queue Representation

Circular Queue

- In a circular queue, the last element points to the first element making a circular link.
- The main advantage of a circular queue over a simple queue is better memory utilization. If the last position is full and the first position is empty, we can insert an element in the first position.

This action is not possible in a simple queue.



Circular Queue Representation

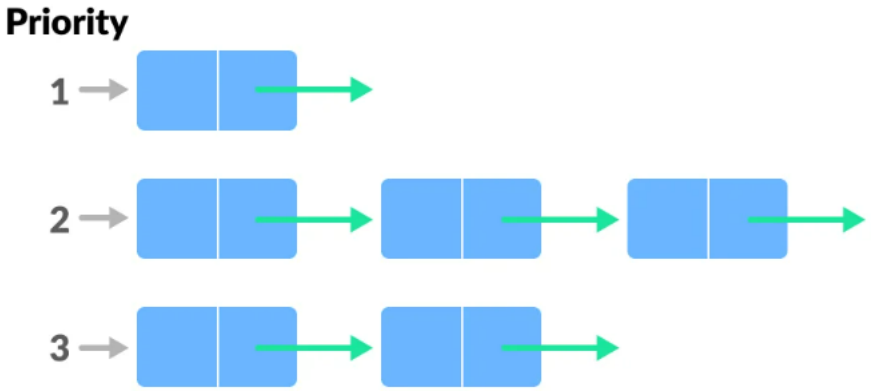
CMC UNIVERSITY

Queues

Priority Queue

A priority queue is a special type of queue in which each element is associated with a priority and is served according to its priority. If elements with the same priority occur, they are served according to their order in the queue.

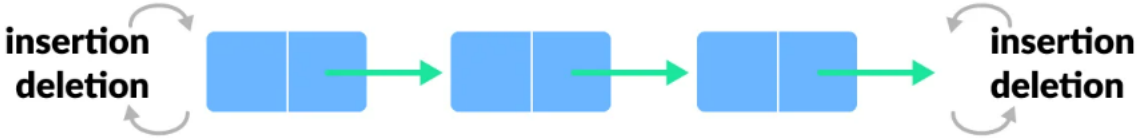
Insertion occurs based on the arrival of the values and removal occurs based on priority.



Priority Queue Representation

Deque (Double Ended Queue)

In a double ended queue, insertion and removal of elements can be performed from either from the front or rear. Thus, it does not follow the FIFO (First In First Out) rule.



Deque Representation


CMC UNIVERSITY
THANK YOU