Data Structures and algorithms

# Link list Data Structure

Godofredo Avena

Nov 21, 2023

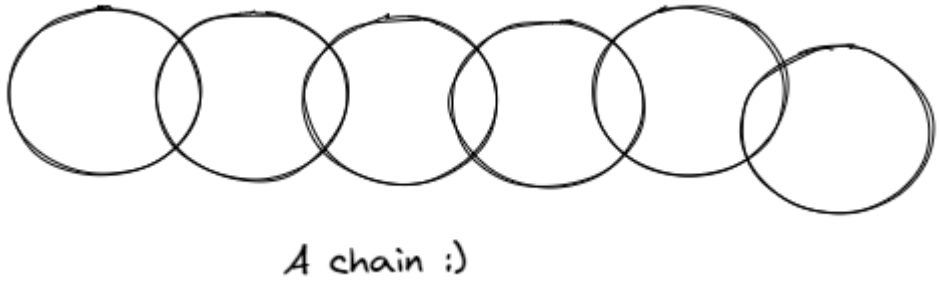
# Algorithms and data structures

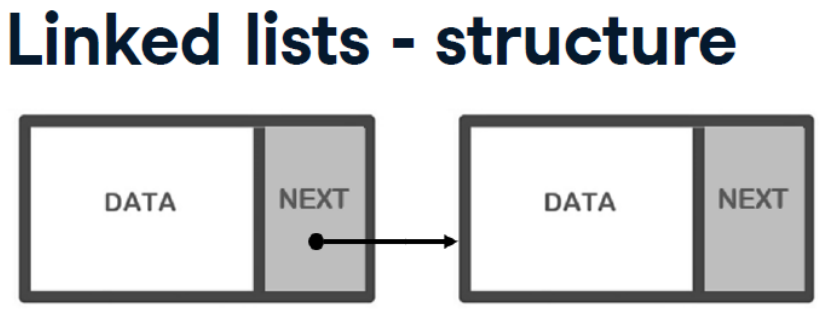
- **Algorithm**: set of instructions that solve a problem
  1. *Design*

  2. *Code*


- **Data structures:** hold and manipulate data when we execute an algorithm
  - **Advanced** data structures: linked lists, stacks, queues...

**Linked Lists** are a data structure that store data in the form of a chain. The structure of a linked list is such that each piece of data has a connection to the next one (and sometimes the previous data as well). Each element in a linked list is called a **node**.



## Advantages of Linked Lists:

1. Because of the chain-like system of linked lists, you can add and remove elements quickly. This also doesn't require reorganizing the data structure unlike arrays or lists. Linear data structures are often easier to implement using linked lists.
2. Linked lists also don't require a fixed size or initial size due to their chainlike structure.

## Disadvantages of a Linked Lists:

1. More memory is required when compared to an array. This is because you need a pointer (which takes up its own memory) to point you to the next element.
2. Search operations on a linked list are very slow. Unlike an array, you don't have the option of random access.

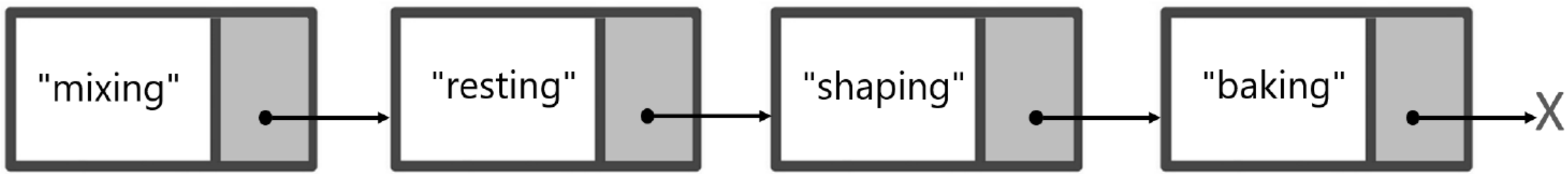
# When Should You Use a Linked List?

You should use a linked list over an array when:

1. You don't know how many items will be in the list (that is one of the advantages - ease of adding items).
2. You don't need random access to any elements (unlike an array, you cannot access an element at a particular index in a linked list).
3. You want to be able to insert items in the middle of the list.
4. You need constant time insertion/deletion from the list (unlike an array, you don't have to shift every other item in the list first).

# Linked lists

bread_steps




- Sequence of data connected through links

# Linked lists - structure

NODE

A Node is a fundamental part of the Linked lists data structure. A node stores two pieces of information:

- **The data element** (which can be a number, string, or any other type of data).
- A **reference or pointer** to the next node in the list (known as next in a singly linked list).

# Linked lists - structure


NODE

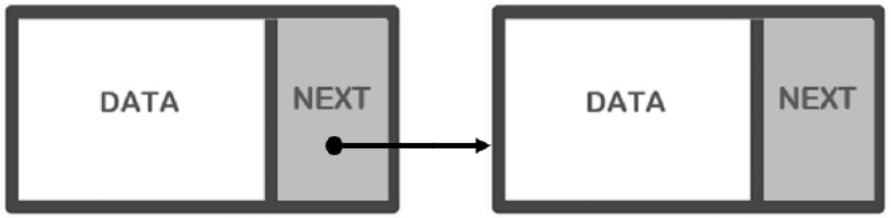
**Data:** This is the actual information that you want to store in the linked list. Each node in a linked list typically holds one data element. For example, if you're creating a linked list of integers, the data in each node would be an integer value.

# Linked lists - structure


NODE

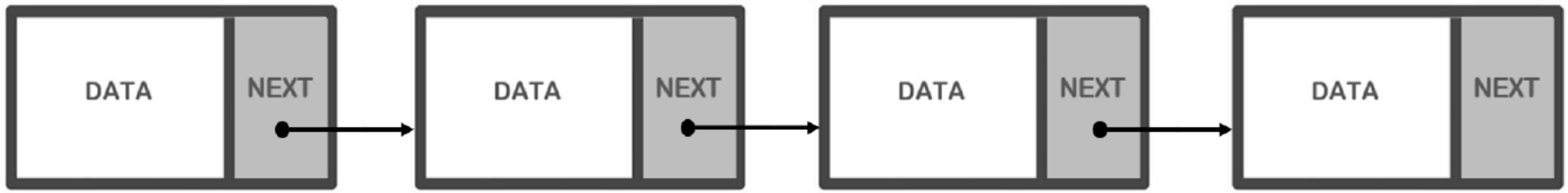
**Next:** This is a pointer or reference in each node that points to the next node in the list. It's what links the nodes together in a sequence. In a singly linked list, each node has a single next pointer. In a doubly linked list, there would also be a previous  pointer to the previous node.

# Linked lists - structure

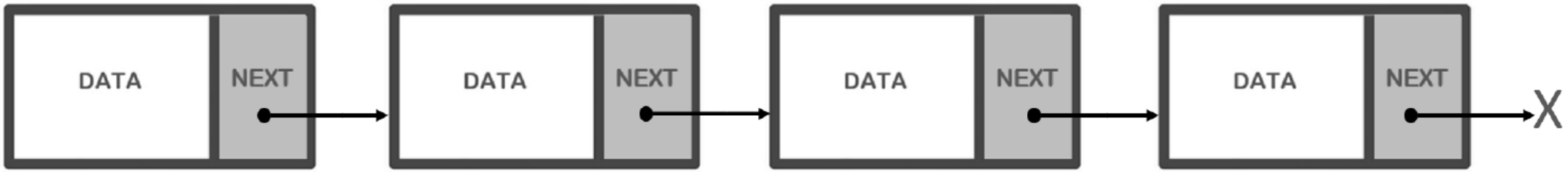



**Next:** This is a pointer or reference in each node that points to the next node in the list. It's what links the nodes together in a sequence. In a singly linked list, each node has a single next pointer. In a doubly linked list, there would also be a previous  pointer to the previous node.

# Linked lists - structure

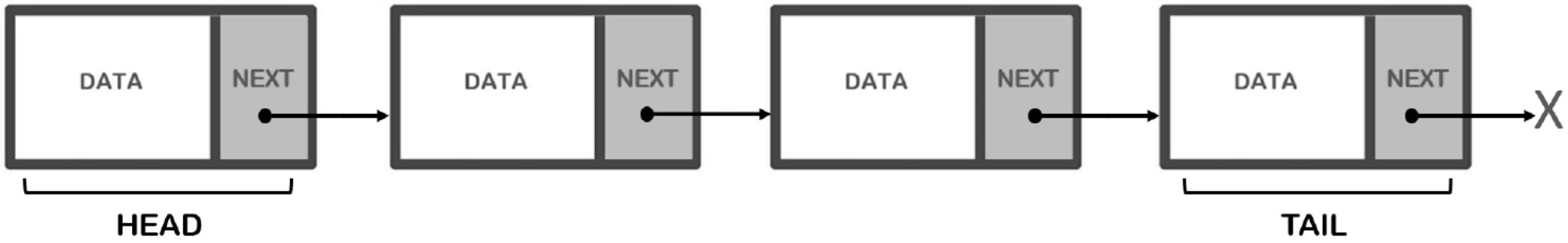


# Linked lists - structure




The last node's "NEXT" section is pointing to a null reference, indicated by a **diagonal cross ('X').** This signifies the end of the list – there are no further nodes to follow.

# Linked lists - structure



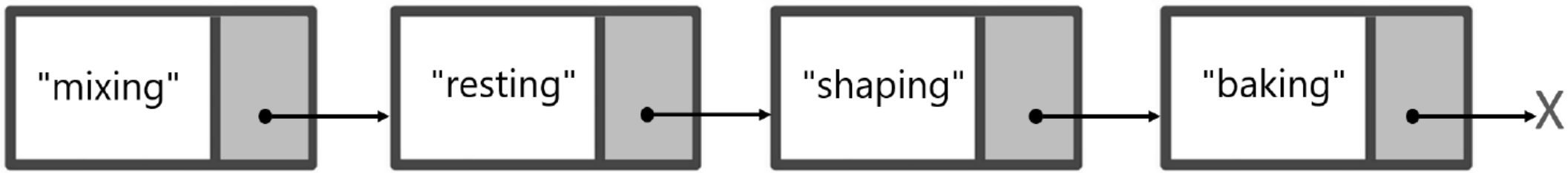

- Data doesn't need to be stored in contiguous blocks of memory
- Data can be located in any available memory address

**Head Node:** This is a reference to the first node in the linked list. The head is used as a starting point to access any node in the list by traversing the next references from the head to the desired node. In an empty list, the head is typically set to None.

**Tail Node:** In the context of a linked list, the tail is the last node in the list. In a singly linked list, the next reference of the tail node is None, indicating the end of the list. In a doubly linked list, the tail also helps in traversing the list backward.

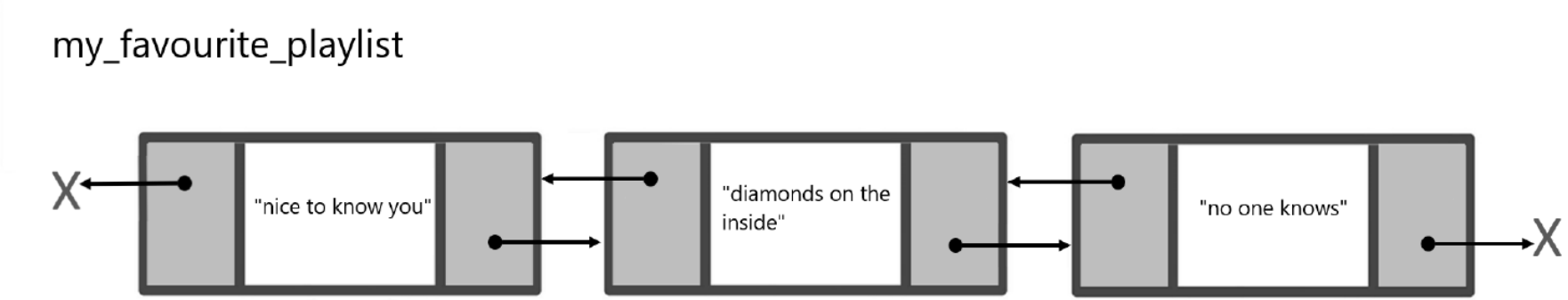
# Singly linked lists

bread_steps




- One link: **singly linked list**

# Doubly linked lists




- Two links in either direction: **doubly linked list**

- The left and right sections of each node have arrows pointing to the next and previous nodes, respectively. This indicates that each node contains references to both the next and the previous nodes in the sequence, which is characteristic of a doubly linked list.

- The leftmost arrow on the first node and the rightmost arrow on the last node point to a null reference, shown by an 'X'. This indicates the ends of the list; there is no previous node before the first node and no next node after the last one.

- The arrows between the nodes show bidirectional connections, meaning you can traverse the list in both directions: forward (to the right) and backward (to the left).

# Linked lists - real uses

- Implement other data structures:
  - stacks
  - queues
  - graphs
- Access information by navigating backward and forward
  - web browser
  - music playlist

# Linked lists - Node class

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

# Linked lists - LinkedList class

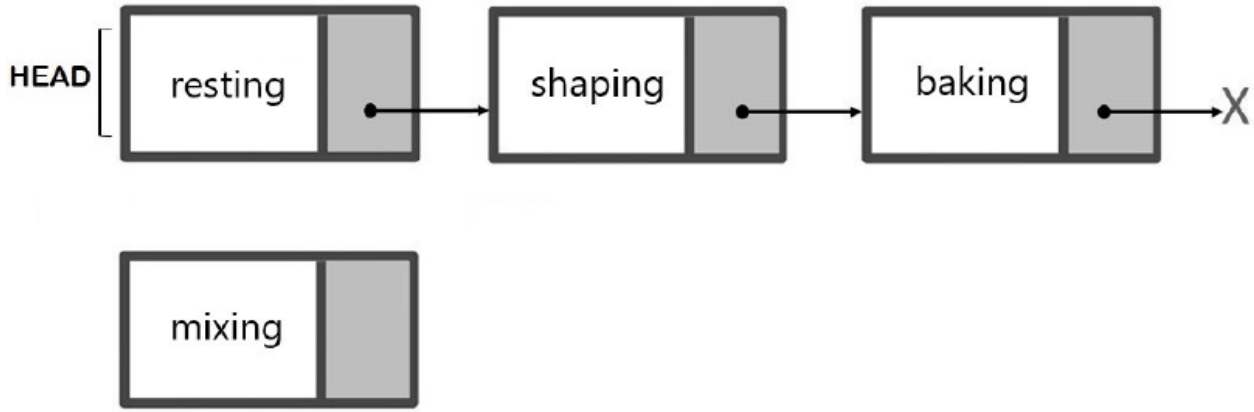
```
class LinkedList:
  def __init__(self):
    self.head = None
    self.tail = None
```

# Linked lists - methods

- `insert_at_beginning()`
- `remove_at_beginning()`
- `insert_at_end()`
- `remove_at_end()`
- `insert_at()`
- `remove_at()`
- `search()`
- …

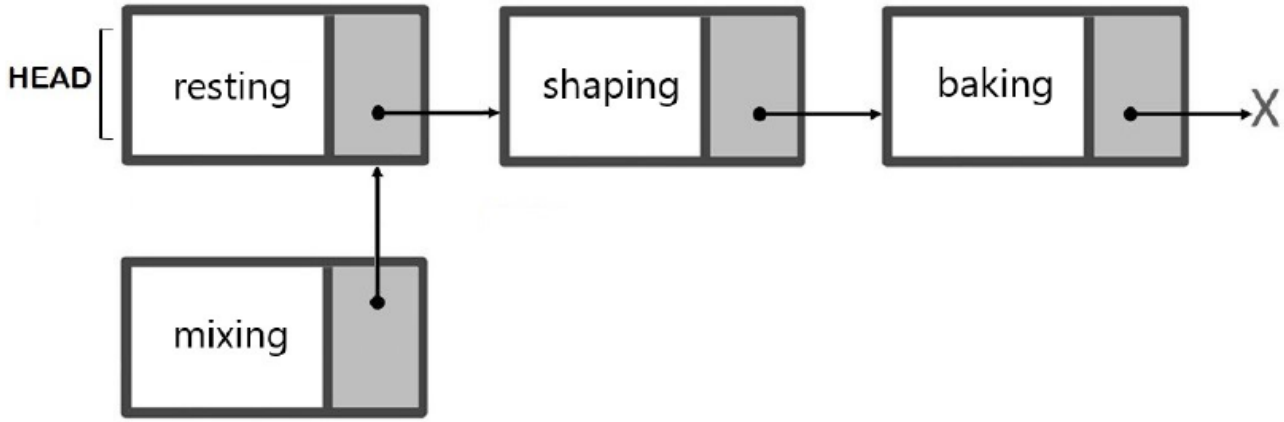
# Linked lists - insert_at_beginning

# Linked lists - insert_at_beginning



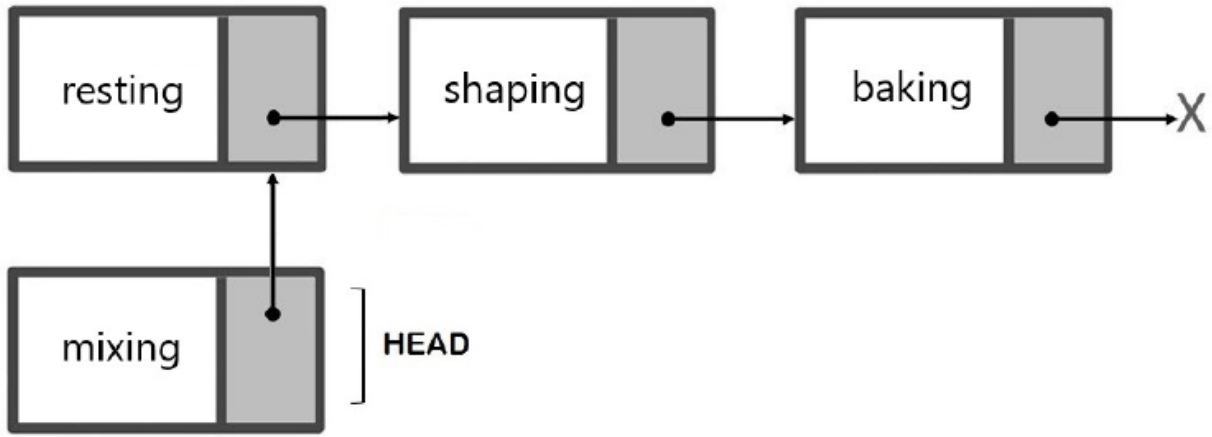

```
def insert_at_beginning(self, data):
    new_node = Node(data)
    if self.head:
```

# Linked lists - insert_at_beginning




```
def insert_at_beginning(self, data):
    new_node = Node(data)
    if self.head:
      new_node.next = self.head
```

# Linked lists - insert_at_beginning




```
def insert_at_beginning(self, data):
    new_node = Node(data)
    if self.head:
      new_node.next = self.head
      self.head = new_node
```

# Linked lists - insert_at_beginning

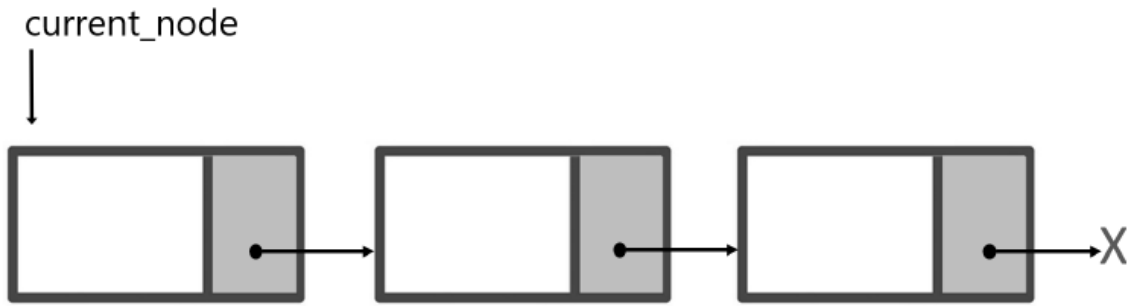

```
def insert_at_beginning(self, data):
    new_node = Node(data)
    if self.head:
      new_node.next = self.head
      self.head = new_node
    else:
      self.tail = new_node
      self.head = new_node
```

# Linked lists - insert_at_end

```
def insert_at_end(self, data):
  new_node = Node(data)
  if self.head:
    self.tail.next = new_node
    self.tail = new_node
  else:
    self.head = new_node
    self.tail = new_node
```

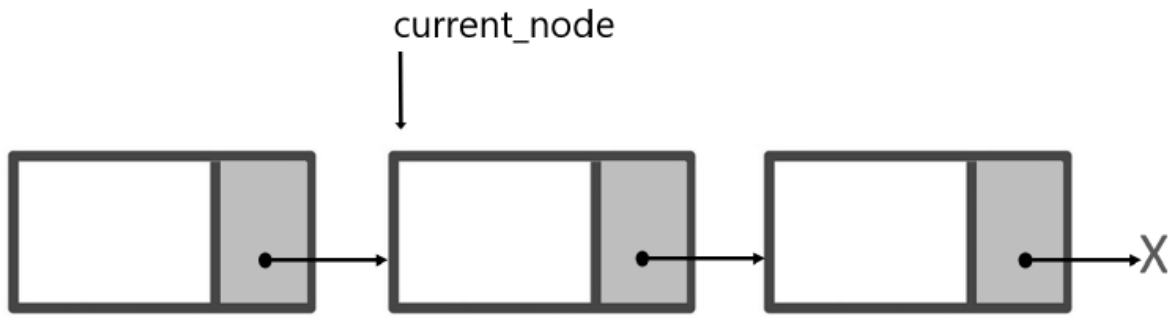
# Linked lists - search

```
def search(self, data):
  current_node = self.head
  while current_node:
    if current_node.data == data:
      return True
```



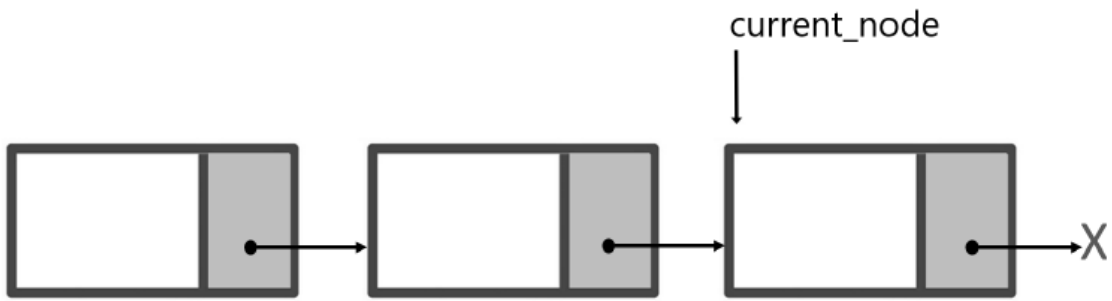
# Linked lists - search

```
def search(self, data):
  current_node = self.head
  while current_node:
    if current_node.data == data:
      return True
    else:
      current_node = current_node.next
```



# Linked lists - search

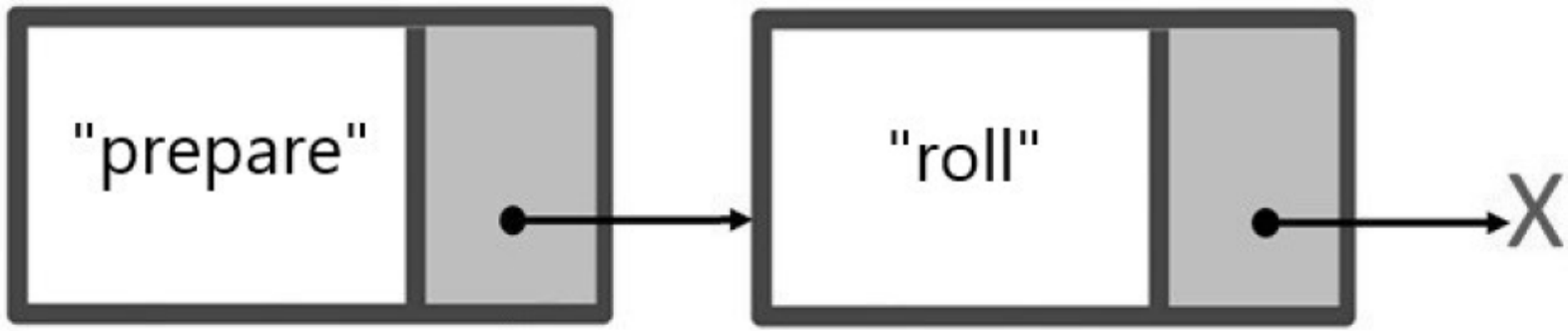
```
def search(self, data):
  current_node = self.head
  while current_node:
    if current_node.data == data:
      return True
    else:
      current_node = current_node.next
  return False
```



# Linked lists - example


```
sushi_preparation = LinkedList()
sushi_preparation.insert_at_end("prepare")
```

# Linked lists - example




```
sushi_preparation = LinkedList()
sushi_preparation.insert_at_end("prepare")
sushi_preparation.insert_at_end("roll")
```

# Linked lists - example


```
sushi_preparation = LinkedList()
sushi_preparation.insert_at_end("prepare")
sushi_preparation.insert_at_end("roll")
sushi_preparation.insert_at_beginning("assemble")
```

# Linked lists - example


```
sushi_preparation.search("roll")
```

```
True
```

```
sushi_preparation.search("mixing")
```

```
False
```

Assignment (to be discuss and pass on Nov 23, 2023):

Update the sample code and create the following methods

**Create the methods**

a. ***remove_beginning(self)***

#this will return the data that was removed at the beginning

b. ***remove_at_end(self)***

#this will return the data that was removed at the end

c. ***remove_at(self,data)***

#this will return the data that was removed else return null if data not found