005 - Lists

A list is a fundamental data structure in programming that represents an ordered collection of elements. It allows you to store and manipulate multiple values of any data type, such as numbers, strings, or even other lists, within a single variable.

Node

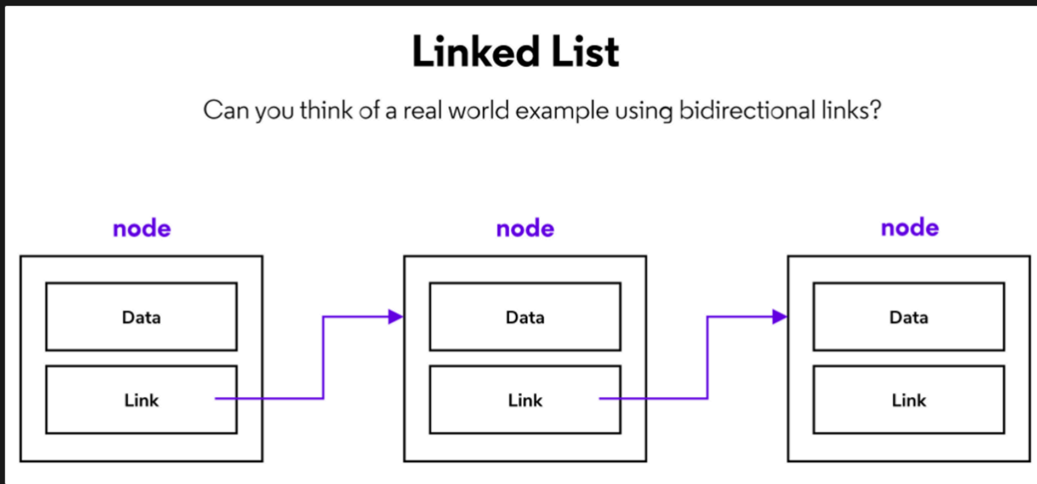
Nodes are the fundamental building blocks of many data structures in Computer Science.

An individual node contains data and links to other nodes.

- The data contained within the nodes can be a variety of types.
- The link or links within the nodes are called **pointers** as they "point" to another node.
- Often, due to the data structure, nodes may only be linked to a single other node.
  - This makes it very important to consider how you implement modifying or removing nodes from a data structure.
  - If you inadvertently remove the single link to a node, that node's data and any linked nodes could be "lost" to your application. When this happens to a node, it is called an *orphaned* node.

```python
class Node:
  def __init__(self, value, link_node=None):
    self.value = value
    self.link_node = link_node

  def set_link_node(self, link_node):
    self.link_node = link_node

  def get_link_node(self):
    return self.link_node

  def get_value(self):
    return self.value

# Add your code below:
yacko = Node("likes to yak")
wacko = Node("has a penchant for hoarding snacks")
dot = Node("enjoys spending time in movie lots")
yacko.set_link_node(dot)
dot.set_link_node(wacko)
dots_data = yacko.get_link_node().get_value()
wackos_data = dot.get_link_node().get_value()
print(dots_data)
print(wackos_data)
```

Linked Lists



- The list is comprised of a series of nodes.
- The head node is the node at the beginning of the list.
- Each node contains data and a link (or pointer) to the next node in the list.
- The list is terminated when a node's link is `null`. This last node is called the tail node.
- Since the nodes use links to denote the next node in the sequence, the nodes are not required to be sequentially located in memory.
- These links also allow for quick insertion and removal of nodes.
- A linked list can typically support some subset of the following operations.
    - push_front - add an item to the front of the linked list

```
def push_front(self, data):
    nn = self.Node(data, self.front)
    if self.front is None:
        self.back = nn
    else:
        self.front.prev= nn
    self.front = nn
```

- **push_back -** add an item to the back of the linked list
- **pop_front -** remove the frontmost item from the linked list

```
def pop_front(self):
    if self.front is not None:
        rm = self.front
        self.front = self.front.next
        if self.front is None:
            self.back = None
        else:
            self.front.prev = None
        del rm
```

- **pop_back -** remove the backmost item from the linked list
- **insert -** given a point within the list insert an item just before that point
- **erase -** remove a node at a specific point within the list
- **erase(a,b) -** erases all nodes between a and b
- **traversals -** some operation that applies to every node in the list

PRINTING A LINKED LIST

```
currNode = ll.front
while currNode:
    print(currNode.value)
    currNode = currNode.next
```

> **A linked lists where each node contains only a single pointer to the next node is called a *singly linked list*.**
>
> **A linked list where each node contains two pointers one to the next node and one to the previous node is called a *doubly linked list*.**

Advantages and Drawbacks of Linked Lists

1. **Advantages:**
    - Linked lists are very easy to grow and shrink as nodes only exist if there is data stored in them. When you grow the linked list, old nodes do not need to be duplicated as part of the grow process.
    - Nodes are only created if there is data to store. No need to preallocate extra space.
    - Data is not stored in consecutive memory locations so a large block of contiguous memory is not required even for storing large amounts of data.
    - Both insertion and removal of any node in the list (assuming that the position of the insertion/removal is known) can be very efficient and runs in constant, O(1) time as it would only require a change of a few pointers. Exactly how long it takes depends on the type of linked list and the exact operations being performed. The key however is that when values are added or removed from a linked list, other values in the list are not moved around.
2. **Drawbacks:**
    - Each piece of data requires the storage of at least one extra pointer. When an array is full, it uses less memory than a linked list of the same size. Furthermore, if the data being stored in each node doesn't require much memory, then the pointer cost relative to the data stored can be significant. For example, an integer takes just as much room to store as a pointer. If you have a singly linked list of integers, the storage of each piece of data is double that of storing it into an array. Thus any array that is more than 50% full is storing more integers with same amount of storage as a singly linked list with the same amount of data.
    - A linked list cannot be searched using binary search as direct access to nodes are not available.
    - Data is not necessarily stored consecutively, this will mean that hardware advantages such as caching won't apply. If your program requires you to do something with all the data in the list, this can have significant impact on performance.

Singly Linked List implementation

```
class Node:
  def __init__(self, value, next_node=None):
    self.value = value
    self.next_node = next_node

  def get_value(self):
    return self.value

  def get_next_node(self):
    return self.next_node

  def set_next_node(self, next_node):
    self.next_node = next_node

class LinkedList:
  def __init__(self, value=None):
    self.head_node = Node(value)

  def get_head_node(self):
    return self.head_node

  def insert_beginning(self, new_value):
    new_node = Node(new_value)
    new_node.set_next_node(self.head_node)
    self.head_node = new_node

  def stringify_list(self):
    string_list = ""
    current_node = self.get_head_node()
    while current_node:
      if current_node.get_value() != None:
        string_list += str(current_node.get_value()) + "\n"
      current_node = current_node.get_next_node()
    return string_list
```

```python
    def remove_node(self, value_to_remove):
      current_node = self.get_head_node()
      if current_node.get_value() == value_to_remove:
        self.head_node = current_node.get_next_node()
      else:
        while current_node:
          next_node = current_node.get_next_node()
          if next_node.get_value() == value_to_remove:
            current_node.set_next_node(next_node.get_next_node())
            current_node = None
          else:
            current_node = next_node
```

Swapping nodes in a Linked List

Given a linked list and the elements to be swapped (`val1` and `val2`), we need to keep track of four values:

- `node1` : the node that matches `val1`
- `node1_prev` : `node1` 's previous node
- `node2` : the node that matches `val2`
- `node2_prev` : `node2` 's previous node

Given an input of a linked list, `val1` , and `val2` , the general steps for doing so is as follows:

1. Iterate through the list looking for the node that matches `val1` to be swapped (`node1`), keeping track of the node's previous node as you iterate (`node1_prev`)
2. Repeat step 1 looking for the node that matches `val2` (giving you `node2` and `node2_prev`)
3. If `node1_prev` is `None` , `node1` was the head of the list, so set the list's head to `node2`
4. Otherwise, set `node1_prev` 's next node to `node2`
5. If `node2_prev` is `None` , set the list's head to `node1`
6. Otherwise, set `node2_prev` 's next node to `node1`
7. Set `node1` 's next node to `node2` 's next node
8. Set `node2` 's next node to `node1` 's next node

```
def swap_nodes(input_list, val1, val2):
    node1 = input_list.head_node
    node2 = input_list.head_node
    # Keeping track of the nodes before our target nodes
    node1_prev = None
    node2_prev = None

    # If both the values are same, there is no point in running the whole fun
ction
    if val1 == val2:
        print("Elements are the same - no swap needed")
        return

    # Setting the previous nodes
    while node1 is not None:
        if node1.get_value() == val1:
            break
        node1_prev = node1
        node1 = node1.get_next_node()

    while node2 is not None:
        if node2.get_value() == val2:
            break
        node2_prev = node2
        node2 = node2.get_next_node()

    # If the nodes are none than the swapping operation is obviously not poss
ible
    if (node1 is None or node2 is None):
        print("Swap not possible - one or more element is not in the list")
        return

    # If the prev node is null, that means it is the first node
    # We assign the second node to the head
    if node1_prev is None:
        input_list.head_node = node2
    else:
        # Otherwise we set the next node for our previous one as the second n
ode
        node1_prev.set_next_node(node2)
```

```
    if node2_prev is None:
        input_list.head_node = node1
    else:
        node2_prev.set_next_node(node1)

    # We store the nect node of the nodes in a temp variable before swapping
them
    temp = node1.get_next_node()
    node1.set_next_node(node2.get_next_node())
    node2.set_next_node(temp)
```

Doubly Linked List Implementation

```
class LinkedList:
    class Node:
        def __init__(self, value, next=None, prev=None):
            self.value = value
            self.next = next
            self.prev = prev

    def __init__(self, front=None, back=None):
        self.front = front
        self.back = back

    def add_to_head(self, value):
        new_node = self.Node(value)
        if self.front is None:
            self.front = new_node
            self.back = new_node
        else:
            new_node.next = self.front
            self.front.prev = new_node
            self.front = new_node
```

```python
    def add_to_tail(self, value):
        new_node = self.Node(value)
        if self.back is None:
            self.front = new_node
            self.back = new_node
        else:
            new_node.prev = self.back
            self.back.next = new_node
            self.back = new_node

    def remove_from_head(self):
        if self.front is None:
            return None
        value = self.front.value
        if self.front == self.back:
            self.front = None
            self.back = None
        else:
            self.front = self.front.next
            self.front.prev = None
        return value

    def remove_from_tail(self):
        if self.back is None:
            return None
        value = self.back.value
        if self.front == self.back:
            self.front = None
            self.back = None
        else:
            self.back = self.back.prev
            self.back.next = None
        return value

    def search(self, value):
        current = self.front
        while current:
            if current.value == value:
                return True
            current = current.next
        return False
```

```python
    def remove(self, value):
        current = self.front
        while current:
            if current.value == value:
                if current == self.front:
                    self.remove_from_head()
                elif current == self.back:
                    self.remove_from_tail()
                else:
                    current.prev.next = current.next
                    current.next.prev = current.prev
                return True
            current = current.next
        return False


# Example usage:

ll = LinkedList()
ll.add_to_head(3)
ll.add_to_head(2)
ll.add_to_head(1)

ll.add_to_tail(4)
ll.add_to_tail(5)

print(ll.search(3))  # Output: True
print(ll.search(6))  # Output: False

ll.remove(3)
ll.remove_from_tail()

currNode = ll.front
while currNode:
    print(currNode.value)
    currNode = currNode.next
```

Implementation using Sentinel Nodes

- Sentinel nodes are nodes that exist at the front and back of a linked list.
- These nodes always exist from the time the linked list is created to the time it is destroyed. They do not hold any data.
- The purpose for their existence is to eliminate most of the special cases when writing functions.
- Most of the special cases in our implementations involve checking whether the front_/back_ pointers or next_/prev_ pointers are nullptr/None at the time or not.
- Sentinel nodes can help us dealing with these situations more easily by preventing these from happening, and let us have our code more simplified.

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

    def __init__(self):
        self.front = self.Node(None)
        self.back = self.Node(None, None, self.front)
        self.front.next = self.back

    def push_front(self, value):
        nn = self.Node(value)
        nn.next = self.front.next
        nn.prev = self.front
        self.front.next.prev = nn
        self.front.next = nn
```

```python
    def push_back(self, value):
        nn = self.Node(value)
        nn.prev = self.back.prev
        nn.next = self.back
        self.back.prev.next = nn
        self.back.prev = nn

    def pop_front(self):
        if self.front.next is not self.back:
            rm = self.front.next
            self.front.next = rm.next
            rm.next.prev = self.front
            del rm

    def pop_back(self):
        if self.back.prev is not self.front:
            rm = self.back.prev
            self.back.prev = rm.prev
            rm.prev.next = self.back
            del rm

    def printLL(self):
        curr = self.front.next
        while curr is not self.back:
            print(curr.data)
            curr = curr.next

ll = LinkedList()
ll.push_front(10)
ll.push_front(20)
ll.push_back(30)
ll.push_back(40)
ll.printLL()
ll.pop_front()
ll.pop_back()
ll.printLL()
```

Pushing an Element to the front

1. Create new node, next node is the node that follows the front sentinel. The previous node is the front sentinel.
2. Make the previous pointer of the node that follows the front sentinel point to the new node
3. Set the next pointer of the front sentinel to the new node.

```
def push_front(self, data):
    nn = self.Node(data, self.front.next, self.front)
    self.front.next.prev = nn
    self.front.next = nn
```

Popping an Element from the front

1. heck to make sure list isn't empty. If it is do nothing. Otherwise continue to next steps. Remember that with sentinels, an empty list still has two nodes (the front and back sentinels). Our empty check is therefore going to look at whether those are the only nodes that exist. We can do this by checking if front sentinel's next_ pointer points to the back sentinel
2. Make a local pointer point to the Node we want to remove. This will be the node that follows the front sentinel as it is the first node with real data. (hold this node so we don't lose it by accident)
3. Make the front sentinel's next pointer point to second data node (the one that follows the one we want to remove
4. Make the previous pointer of the node that now follows the front sentinel point back to the front sentinel

```
def pop_front(self):
    if self.front.next is not self.back:
        rm = self.front.next
        rm.next.prev = rm.prev
        rm.prev.next = rm.next
        del rm
```

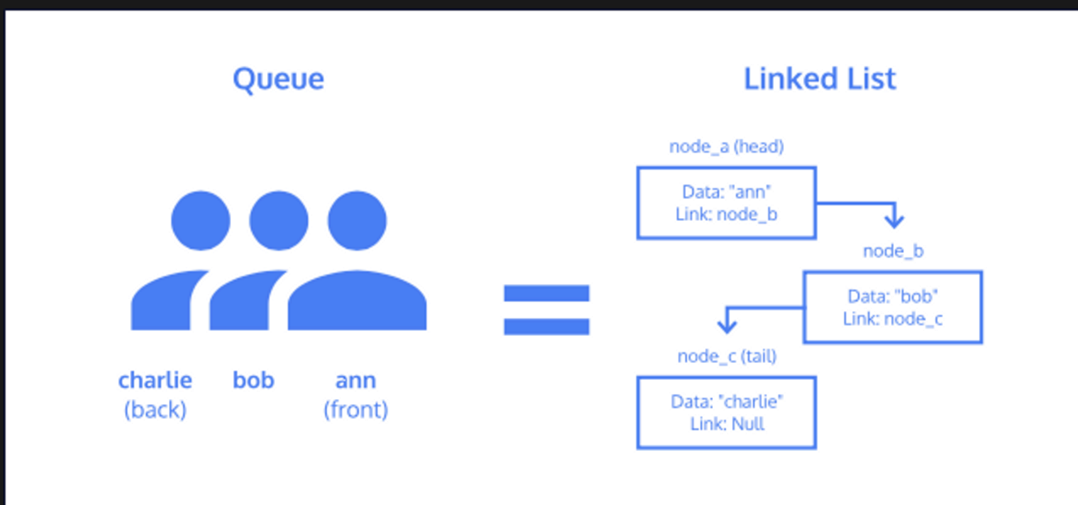
Queues

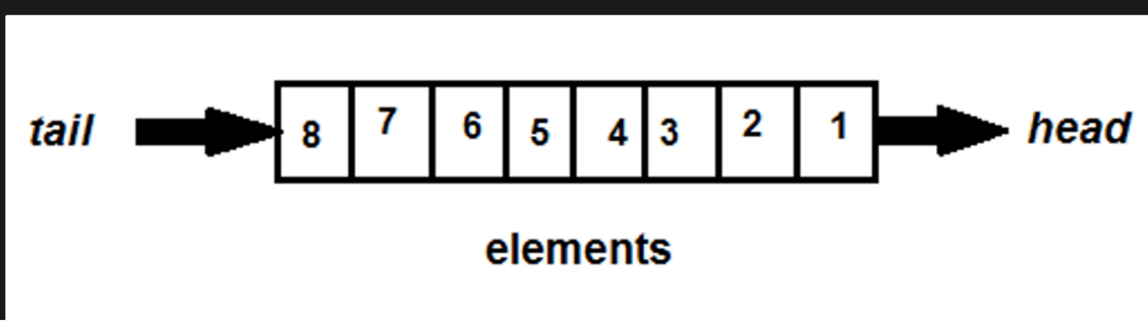
A queue is a data structure which contains an ordered set of data.

Queues provide three methods for interaction:

- **Enqueue -** adds data to the "back" or end of the queue
- **Dequeue -** provides and removes data from the "front" or beginning of the queue
- **Peek -** reveals data from the "front" of the queue without removing it

This data structure mimics a physical queue of objects like a line of people buying movie tickets. Each person has a name (the data). The first person to *enqueue*, or get into line, is both at the front and back of the line. As each new person enqueues, they become the new back of the line.





💡 **Queues are FIFO (First In First Out) Structure**

One last constraint that may be placed on a queue is its length. If a queue has a limit on the amount of data that can be placed into it, it is considered a *bounded queue*.

Similar to stacks, attempting to enqueue data onto an already full queue will result in a *queue overflow*. If you attempt to dequeue data from an empty queue, it will result in a *queue underflow*.

```
class Node:
  def __init__(self, value, next_node=None):
    self.value = value
    self.next_node = next_node

  def set_next_node(self, next_node):
    self.next_node = next_node

  def get_next_node(self):
    return self.next_node

  def get_value(self):
    return self.value

from node import Node

from node import Node

class Queue:
  def __init__(self, max_size=None):
    self.head = None
    self.tail = None
    self.max_size = max_size
    self.size = 0
```

```
  def enqueue(self, value):
    if self.has_space():
      item_to_add = Node(value)
      print("Adding " + str(item_to_add.get_value()) + " to the queue!")
      if self.is_empty():
        self.head = item_to_add
        self.tail = item_to_add
      else:
        self.tail.set_next_node(item_to_add)
        self.tail = item_to_add
      self.size += 1
    else:
      print("Sorry, no more room!")

  # Add your dequeue method below:
  def dequeue(self):
    if (not self.is_empty()):
      item_to_remove = self.head
      print("Removing " + str(item_to_remove.get_value()) + " from the queu
e!")
      if (self.size == 1):
        self.head, self.tail = None, None
      else:
        self.head = self.head.get_next_node()
      self.size -= 1
      return item_to_remove.get_value()
    else:
      print("This queue is totally empty!")

  def peek(self):
    if self.is_empty():
      print("Nothing to see here!")
    else:
      return self.head.get_value()

  def get_size(self):
    return self.size
```

```python
    def has_space(self):
      if self.max_size == None:
        return True
      else:
        return self.max_size > self.get_size()

    def is_empty(self):
      return self.size == 0
```

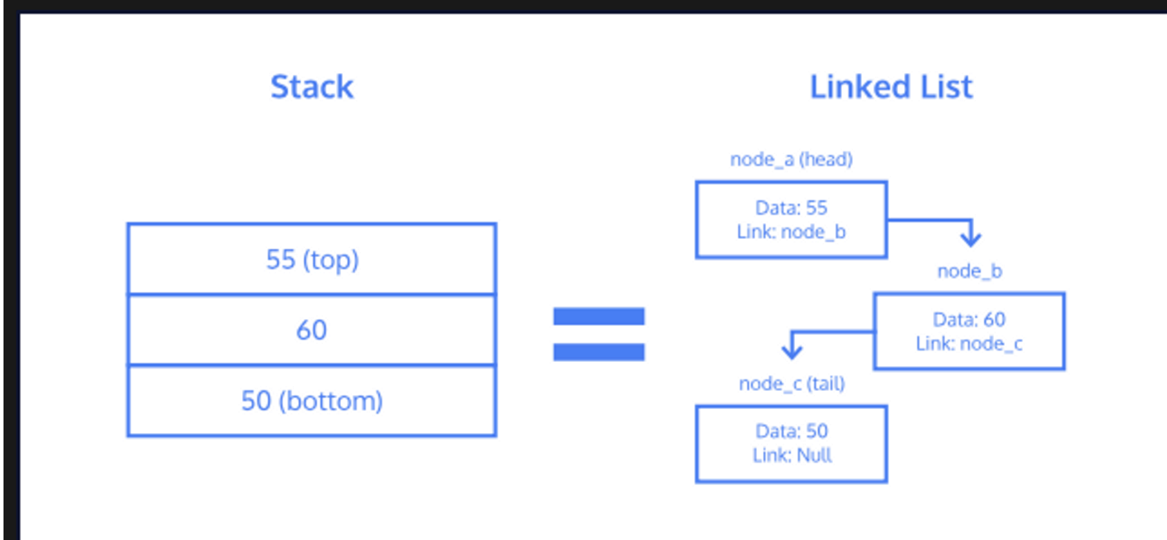
Stacks

A stack is a data structure which contains an ordered set of data.

Stacks provide three methods for interaction:

- **Push -** adds data to the "top" of the stack
- **Pop -** returns and removes data from the "top" of the stack
- **Peek -** returns data from the "top" of the stack without removing it

Stacks mimic a physical "stack" of objects. Consider a set of gym weights. Attempting to push data onto an already full stack will result in a *stack overflow*. Similarly, if you attempt to pop data from an empty stack, it will result in a *stack underflow*.



💡 **Stacks are LIFO (last In First Out) Structure**

```
class Node:
  def __init__(self, value, next_node=None):
    self.value = value
    self.next_node = next_node

  def set_next_node(self, next_node):
    self.next_node = next_node

  def get_next_node(self):
    return self.next_node

  def get_value(self):
    return self.value

from node import Node

class Stack:
  def __init__(self, limit=1000):
    self.top_item = None
    self.size = 0
    self.limit = limit

  def push(self, value):
    if self.has_space():
      item = Node(value)
      item.set_next_node(self.top_item)
      self.top_item = item
      self.size += 1
      print("Adding {} to the pizza stack!".format(value))
    else:
      print("No room for {}!".format(value))
```

```python
def pop(self):
  if not self.is_empty():
    item_to_remove = self.top_item
    self.top_item = item_to_remove.get_next_node()
    self.size -= 1
    print("Delivering " + item_to_remove.get_value())
    return item_to_remove.get_value()
  print("All out of pizza.")

def peek(self):
  if not self.is_empty():
    return self.top_item.get_value()
  print("Nothing to see here!")

def has_space(self):
  return self.limit > self.size

def is_empty(self):
  return self.size == 0
```