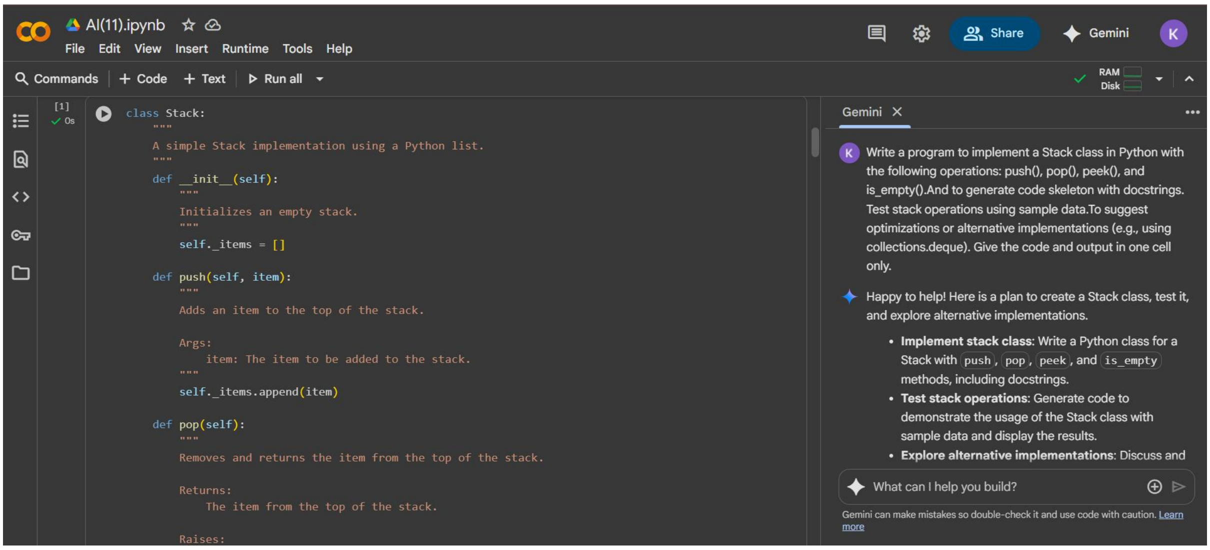
Assignment-11.4

Task 1:

Write a program to implement a Stack class in Python with the following operations: push(), pop(), peek(), and is_empty().And to generate code skeleton with docstrings. Test stack operations using sample data.To suggest optimizations or alternative implementations (e.g., using collections.deque).

Code and Output:




```
        Raises:
            IndexError: If the stack is empty.
        """
        if not self._items:
            raise IndexError("pop from empty stack")
        return self._items.pop()

    def peek(self):
        """
        Returns the item from the top of the stack without removing it.

        Returns:
            The item from the top of the stack.

        Raises:
            IndexError: If the stack is empty.
        """
        if not self._items:
            raise IndexError("peek from empty stack")
        return self._items[-1]

    def is_empty(self):
        """
        Checks if the stack is empty.

        Returns:
            True if the stack is empty, False otherwise.
        """
        return not self._items
```

```python
# 1. Create an instance of the Stack class.
stack = Stack()

# 2. Push a few sample items onto the stack using the push method.
stack.push("apple")
stack.push("banana")
stack.push("cherry")

# 3. Check if the stack is empty using the is_empty method and print the result.
print(f"Is the stack empty? {stack.is_empty()}")

# 4. Peek at the top item of the stack using the peek method and print the result.
print(f"Top item on the stack: {stack.peek()}")

# 5. Pop a few items from the stack using the pop method and print each popped item.
print("Popping items from the stack:")
print(stack.pop())
print(stack.pop())

# 6. Check if the stack is empty again and print the result.
print(f"Is the stack empty after popping? {stack.is_empty()}")

# 7. Attempt to pop from the empty stack to demonstrate the IndexError handling and print a message i
print("Attempting to pop from an empty stack:")
try:
    print(stack.pop())
except IndexError as e:
    print(f"Caught expected error: {e}")
```

```
Is the stack empty? False
Top item on the stack: cherry
Popping items from the stack:
cherry
banana
Is the stack empty after popping? False
Attempting to pop from an empty stack:
apple
```

Reasoning: Discuss alternative stack implementations, specifically `collections.deque`, explain its advantages, and provide a code example for a deque-based stack implementation with docstrings and comments.

```python
import collections

# Discuss alternative implementations
print("Alternative Stack Implementations:")
print("While a Python list can be used to implement a stack, operations like `pop(0)` (removing from the beginning) and `insert(0, item)` (inserting at the beginning) are inefficient because they require shifting all subsequent elements.")
print("A more efficient alternative for implementing a stack (where operations occur at one end, like the end of the list) is `collections.deque` (double-ended queue).")
print("`collections.deque` is optimized for appending and popping elements from both ends, making it suitable for both stacks (LIFO - Last-In, First-Out) and queues (FIFO - First-In, First-Out).")
print("Using `append()` and `pop()` with a `deque` for stack operations is O(1) on average, which is more efficient than using `insert(0, item)` or `pop(0)` with a list for operations on the 'front' of the list.")
print("-" * 30)

# Provide a code example using collections.deque
class DequeStack:
    """
    A Stack implementation using collections.deque.
    """
    def __init__(self):
        """
        Initializes an empty stack using a deque.
        """
        self._items = collections.deque()

    def push(self, item):
        """
        Adds an item to the top of the stack.

        Args:
            item: The item to be added to the stack.
        """
        # Use append to add to the right end (top of the stack)
        self._items.append(item)

    def pop(self):
        """
        Removes and returns the item from the top of the stack.

        Returns:
            The item from the top of the stack.
```

```python
        Raises:
            IndexError: If the stack is empty.
        """
        # Use pop to remove from the right end (top of the stack)
        if not self._items:
            raise IndexError("pop from empty stack")
        return self._items.pop()

    def peek(self):
        """
        Returns the item from the top of the stack without removing it.

        Returns:
            The item from the top of the stack.

        Raises:
            IndexError: If the stack is empty.
        """
        # Access the last element (top of the stack)
        if not self._items:
            raise IndexError("peek from empty stack")
        return self._items[-1]

    def is_empty(self):
        """
        Checks if the stack is empty.

        Returns:
            True if the stack is empty, False otherwise.
        """
        # Check if the deque is empty by checking its length
        return len(self._items) == 0
```

```
Alternative Stack Implementations:
While a Python list can be used to implement a stack, operations like `pop(0)` (removing from the beginning) and `insert(0, item)` (inserting at the beginning) are inefficient because they require shifting all subsequent elements.
A more efficient alternative for implementing a stack (where operations occur at one end, like the end of the list) is `collections.deque` (double-ended queue).
`collections.deque` is optimized for appending and popping elements from both ends, making it suitable for both stacks (LIFO - Last-In, First-Out) and queues (FIFO - First-In, First-Out).
Using `append()` and `pop()` with a `deque` for stack operations is O(1) on average, which is more efficient than using `insert(0, item)` or `pop(0)` with a list for operations on the 'front' of the list.
------------------------------
```

Reasoning: Based on the previous subtasks, the current stack implementation uses a Python list. While simple, list operations like inserting or deleting at the beginning can be inefficient (O(n)). For a stack, we are primarily concerned with operations at one end (the "top"). Using `append()` and `pop()` on a list operates on the end, which is efficient (O(1) on average). However, the prompt specifically asks for *optimizations* and suggests `collections.deque`. Although the current list implementation using `append` and `pop` is already efficient for stack operations, the prompt implies exploring alternatives that *might* be more performant in certain scenarios or are generally considered better for double-ended operations (which `deque` is designed for). Therefore, suggesting `collections.deque` as an alternative is appropriate as it guarantees O(1) for append and pop at either end, which is a potential optimization compared to list operations at the *front* if the stack were implemented that way (though our current list implementation is already optimized by using the end). Additionally, for very large numbers of operations, `deque` might have slightly better constant factors due to its underlying implementation.

```python
print("Possible Optimizations and Alternative Implementations:")
print("\nThe current Stack implementation uses a Python list, performing push operations with `append()` and pop/peek operations with `pop()` and indexing `[-1]` respectively. These operations on the end of a list are generally efficient, with an average time complexity of O(1).")

print("\nHowever, for scenarios involving a very large number of push and pop operations, or if the stack were implemented using operations on the *beginning* of the list (e.g., `insert(0, item)` and `pop(0)`, which have O(n) complexity), performance could degrade.")

print("\nA common and highly recommended optimization/alternative implementation for a stack in Python is to use `collections.deque` (double-ended queue).")

print("\nWhy `collections.deque` is beneficial:")
print("- `collections.deque` is specifically designed for efficient appending and popping from both ends of the sequence.")
print("- Both `append()` (for push) and `pop()` (for pop) operations on a `deque` have a guaranteed time complexity of O(1). This is more consistent than list's amortized O(1) append, and significantly better than list's O(n) for operations on the front.")
print("- While the current list implementation using `append` and `pop` at the end is already efficient (O(1) on average), `deque` offers guaranteed O(1) for these operations and is the preferred data structure when you know you'll be primarily adding/removing from the ends.")
print("- For very high-performance or concurrent applications, `deque`'s underlying implementation might offer slight advantages.")

print("\nIn summary, while the current list-based implementation using end operations is efficient, `collections.deque` is a robust and slightly more predictable alternative, guaranteeing O(1) for stack operations (push and pop) and is the idiomatic choice for such use cases in Python.")
```

```
Possible Optimizations and Alternative Implementations:

The current Stack implementation uses a Python list, performing push operations with `append()` and pop/peek operations with `pop()` and indexing `[-1]` respectively. These operations on the end of a list are generally efficient, with an average time complexity of O(1).

However, for scenarios involving a very large number of push and pop operations, or if the stack were implemented using operations on the *beginning* of the list (e.g., `insert(0, item)` and `pop(0)`, which have O(n) complexity), performance could degrade.

A common and highly recommended optimization/alternative implementation for a stack in Python is to use `collections.deque` (double-ended queue).

Why `collections.deque` is beneficial:
- `collections.deque` is specifically designed for efficient appending and popping from both ends of the sequence.
- Both `append()` (for push) and `pop()` (for pop) operations on a `deque` have a guaranteed time complexity of O(1). This is more consistent than list's amortized O(1) append, and significantly better than list's O(n) for operations on the front.
- While the current list implementation using `append` and `pop` at the end is already efficient (O(1) on average), `deque` offers guaranteed O(1) for these operations and is the preferred data structure when you know you'll be primarily adding/removing from the ends.
- For very high-performance or concurrent applications, `deque`'s underlying implementation might offer slight advantages.

In summary, while the current list-based implementation using end operations is efficient, `collections.deque` is a robust and slightly more predictable alternative, guaranteeing O(1) for stack operations (push and pop) and is the idiomatic choice for such use cases in Python.
```

Explanation:

Notebook Gemini

Certainly! Here's an explanation of the code cells in this notebook:

1. **Cell 1 (Code Cell - `632a92f3`):** This cell defines the `Stack` class.
   - It uses a Python list (`self._items`) to store the elements of the stack.
   - The `__init__` method initializes an empty list when a new `Stack` object is created.
   - `push(item)` adds an `item` to the end of the list, effectively putting it on top of the stack.
   - `pop()` removes and returns the last item from the list (the top of the stack). It includes error handling to raise an `IndexError` if the stack is empty.
   - `peek()` returns the last item of the list (the top of the stack) without removing it. It also includes error handling for an empty stack.
   - `is_empty()` checks if the list is empty and returns `True` if it is, indicating the stack is empty.
2. **Cell 2 (Code Cell - `d5ee5a2e`):** This cell tests the functionality of the `Stack` class.
   - It creates an instance of the `Stack`.
   - It tests `is_empty()` on the newly created empty stack.
   - It pushes elements (10, 20, 30) onto the stack.
   - It tests `is_empty()` again after pushing elements.
   - It uses `peek()` to see the top element (30).
   - It uses a `while` loop and `pop()` to remove and print elements until the stack is empty, demonstrating the LIFO (Last-In, First-Out) behavior.
   - It tests `is_empty()` after all elements are popped.
   - Finally, it includes `try...except` blocks to show how the `IndexError` is caught when attempting to `pop()` or `peek()` from an empty stack.
3. **Cell 3 (Markdown Cell - `eeaaf08d`):** This cell is a markdown cell that serves as a header for the suggestion of optimizations.
4. **Cell 4 (Markdown Cell - `d8b4ecf1`):** This markdown cell provides a summary of the key findings from implementing and testing the Stack class. It also suggests potential optimizations and next steps,

Task 2:

Write a python program to Implement a Queue with enqueue(), dequeue(), and is_empty() methods. First, implement using Python lists.Then, review performance and suggest a more efficient implementation (using collections.deque).

Code and Output:

Untitled8.ipynb

File Edit View Insert Runtime Tools Help

Commands + Code + Text Run all

[7] 0s

```python
# Implement queue with python lists
class ListQueue:
    def __init__(self):
        self._items = []

    def enqueue(self, item):
        self._items.append(item)

    def dequeue(self):
        if not self.is_empty():
            return self._items.pop(0)
        else:
            return None

    def is_empty(self):
        return len(self._items) == 0

# Test list-based queue
list_queue = ListQueue()

print("Adding elements to the list-based queue:")
list_queue.enqueue(10)
list_queue.enqueue(20)
list_queue.enqueue(30)
print("List-based queue after adding elements.")

print("\nIs the list-based queue empty?", list_queue.is_empty())

print("\nDequeuing elements from list-based queue:")
while not list_queue.is_empty():
```

Share Gemini RAM Disk

Gemini

write a python program to Implement a Queue with enqueue(), dequeue(), and is_empty() methods. First, implement using Python lists.Then, review performance and suggest a more efficient implementation (using collections.deque).

Certainly! Here is a plan to implement a queue using Python lists and then suggest a more efficient implementation using `collections.deque`.

- **Implement queue with python lists**: Create a class for a queue using Python lists, including `enqueue()`, `dequeue()`, and `is_empty()` methods.
- **Test list-based queue**: Write code to test the implemented queue with basic operations.
- **Analyze performance of list-based queue**: Discuss the time complexity of the list-based implementation.
- **Implement queue with `collections.deque`**: Create a class for a queue using `collections.deque`, including `enqueue()`, `dequeue()`, and `is_empty()` methods.
- **Test `collections.deque`-based queue**: Write code to test the implemented queue with basic operations.
- **Analyze performance of `collections.deque`-based queue**: Discuss the time complexity of the `collections.deque`-based

What can I help you build?

Gemini can make mistakes so double-check it and use code with caution. Learn more

Commands + Code + Text ▷ Run all

[7] ✓ 0s

```
print("\nDequeuing elements from list-based queue:")
while not list_queue.is_empty():
    dequeued_item = list_queue.dequeue()
    print(f"Dequeued: {dequeued_item}")

print("\nIs the list-based queue empty after dequeuing all elements?", list_queue.is_empty())

print("\nAttempting to dequeue from an empty list-based queue:")
dequeued_item_empty = list_queue.dequeue()
print(f"Dequeued from empty list-based queue: {dequeued_item_empty}")

# Analyze performance of list-based queue
print("\n--- List-based Queue Performance ---")
print("Time Complexity of enqueue(): O(1) (average)")
print("Time Complexity of dequeue(): O(n)")
print("Time Complexity of is_empty(): O(1)")
print("Summary of performance:")
print("The list-based queue has O(1) enqueue and is_empty operations. The dequeue operation is the bottlen


# Implement queue with collections.deque
from collections import deque

class DequeQueue:
    def __init__(self):
        self._items = deque()

    def enqueue(self, item):
        self._items.append(item)
```

```
    def enqueue(self, item):
        self._items.append(item)

    def dequeue(self):
        if not self.is_empty():
            return self._items.popleft()
        else:
            return None

    def is_empty(self):
        return len(self._items) == 0

# Test collections.deque-based queue
deque_queue = DequeQueue()

print("\n--- collections.deque-based Queue ---")
print("Adding elements to the deque-based queue:")
deque_queue.enqueue(100)
deque_queue.enqueue(200)
deque_queue.enqueue(300)
print("Deque-based queue after adding elements.")

print("\nIs the deque-based queue empty?", deque_queue.is_empty())

print("\nDequeuing elements from deque-based queue:")
while not deque_queue.is_empty():
    dequeued_item = deque_queue.dequeue()
    print(f"Dequeued: {dequeued_item}")

print("\nIs the deque-based queue empty after dequeuing all elements?", deque_queue.is_empty())
```

```python
print("\nAttempting to dequeue from an empty deque-based queue:")
dequeued_item_empty = deque_queue.dequeue()
print(f"Dequeued from empty deque-based queue: {dequeued_item_empty}")

# Analyze performance of collections.deque-based queue
print("\n--- collections.deque-based Queue Performance ---")
print("Time Complexity of enqueue(): O(1)")
print("Time Complexity of dequeue(): O(1)")
print("Time Complexity of is_empty(): O(1)")
print("Summary: All key operations (enqueue, dequeue, is_empty) for the deque-based queue are typically O(1).")

# Compare performance
print("\n--- Performance Comparison ---")
print("Comparison with List-based Queue:")
print("  enqueue(): Deque is O(1), List is O(1) (average). Similar performance.")
print("  dequeue(): Deque is O(1), List is O(n). Deque is significantly faster for dequeue, especially for large queues.")
print("  is_empty(): Deque is O(1), List is O(1). Similar performance.")

print("\nConclusion:")
print("The collections.deque-based implementation is generally more efficient for queue operations compared to the list-based implementation.")
print("This is primarily because both adding (enqueue) and removing from the front (dequeue) operations have a consistent O(1) time complexity with deque, whereas the list-based dequeue has an O(n) complexity.")
print("For large queues, the performance difference in the dequeue operation makes the deque-based implementation the preferred choice.")
```

```
Adding elements to the list-based queue:
List-based queue after adding elements.

Is the list-based queue empty? False

Dequeuing elements from list-based queue:
Dequeued: 10
Dequeued: 20
Dequeued: 30

Is the list-based queue empty after dequeuing all elements? True

Attempting to dequeue from an empty list-based queue:
Dequeued from empty list-based queue: None

--- List-based Queue Performance ---
Time Complexity of enqueue(): O(1) (average)
Time Complexity of dequeue(): O(n)
Time Complexity of is_empty(): O(1)
Summary of performance:
The list-based queue has O(1) enqueue and is_empty operations. The dequeue operation is the bottleneck with O(n) time complexity due to element shifting.

--- collections.deque-based Queue ---
Adding elements to the deque-based queue:
Deque-based queue after adding elements.

Is the deque-based queue empty? False
```

```
Dequeuing elements from deque-based queue:
Dequeued: 100
Dequeued: 200
Dequeued: 300

Is the deque-based queue empty after dequeuing all elements? True

Attempting to dequeue from an empty deque-based queue:
Dequeued from empty deque-based queue: None

--- collections.deque-based Queue Performance ---
Time Complexity of enqueue(): O(1)
Time Complexity of dequeue(): O(1)
Time Complexity of is_empty(): O(1)
Summary: All key operations (enqueue, dequeue, is_empty) for the deque-based queue are typically O(1).

--- Performance Comparison ---
Comparison with List-based Queue:
  enqueue(): Deque is O(1), List is O(1) (average). Similar performance.
  dequeue(): Deque is O(1), List is O(n). Deque is significantly faster for dequeue, especially for large queues.
  is_empty(): Deque is O(1), List is O(1). Similar performance.

Conclusion:
The collections.deque-based implementation is generally more efficient for queue operations compared to the list-based implementation.
This is primarily because both adding (enqueue) and removing from the front (dequeue) operations have a consistent O(1) time complexity with deque, whereas the list-based dequeue has an O(n) complexity.
For large queues, the performance difference in the dequeue operation makes the deque-based implementation the preferred choice.
```

Explanation:

K explanation

Certainly! The code in cell `62142fd8` implements and compares two ways of creating a queue in Python: one using a standard Python list and another using the `collections.deque` object.

Here's a breakdown:

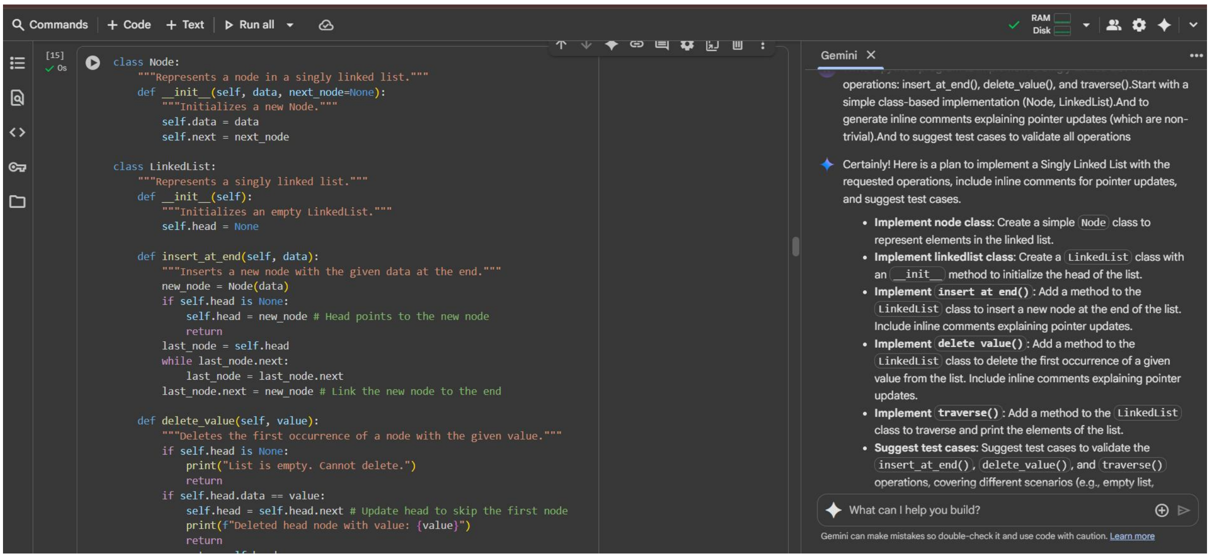
1. **List-based Queue Implementation (`ListQueue` class):**
   - It uses a Python list (`self._items`) to store the queue elements.
   - `enqueue()` : Adds an item to the *end* of the list using `append()` , which is generally an efficient O(1) operation.
   - `dequeue()` : Removes an item from the *beginning* of the list using `pop(0)` . This is an inefficient O(n) operation because all subsequent elements need to be shifted to the left.
   - `is_empty()` : Checks if the list is empty using `len()` , which is an O(1) operation.
2. **Testing the List-based Queue:**
   - This section creates an instance of `ListQueue` , adds elements, checks if it's empty, removes elements one by one, and then attempts to dequeue from an empty queue to show how it handles that case (returning `None`).
3. **Performance Analysis of List-based Queue:**
   - This part explicitly states the time complexity for each operation (`enqueue` as O(1) average, `dequeue` as O(n), and `is_empty` as O(1)). It highlights that `dequeue` is the bottleneck.
4. **`collections.deque` -based Queue Implementation (`DequeQueue` class):**
   - It uses a `collections.deque` object (`self._items`) to store the queue elements. Deques are designed for efficient appending and popping from both ends.
   - `enqueue()` : Adds an item to the *right* end of the deque using `append()` , which is an O(1) operation.
   - `dequeue()` : Removes an item from the *left* end of the deque using `popleft()` . This is an efficient O(1) operation because deques are implemented as doubly linked lists internally, allowing constant time removal from either end.
   - `is_empty()` : Checks if the deque is empty using `len()` , which is an O(1) operation.
5. **Testing the `collections.deque` -based Queue:**
   - Similar to the list-based test, this section creates a `DequeQueue` , adds elements, checks for emptiness, dequeues elements, and tests dequeuing from an empty queue.
6. **Performance Analysis of `collections.deque` -based Queue:**
   - This part states the time complexity for the deque operations, showing that all (`enqueue` , `dequeue` , `is_empty`) are O(1).
7. **Performance Comparison and Conclusion:**
   - This section directly compares the time complexities of the two implementations, emphasizing that the `collections.deque` is significantly more efficient for the `dequeue` operation (O(1) vs O(n) for the list).
   - The conclusion recommends using `collections.deque` for implementing queues in Python, especially when dealing with large queues, due to its consistent O(1) performance for all key operations.

In summary, the code demonstrates that while a Python list *can* be used to implement a queue, the `collections.deque` provides a much more performant solution, particularly for the dequeue operation, which is a fundamental part of queue behavior.

Task 3:

Write a python program to Implement a Singly Linked List with operations: insert_at_end(), delete_value(), and traverse().Start with a simple class-based implementation (Node, LinkedList).And to generate inline comments explaining pointer updates (which are non-trivial).And to suggest test cases to validate all operations.

Code and Output:



```python
        current = self.head
        previous = None
        while current and current.data != value:
            previous = current
            current = current.next
        if current is None:
            print(f"Value {value} not found in the list.")
            return
        previous.next = current.next # Link previous node to the node after current, bypassing current

    def traverse(self):
        """Traverses the list and prints elements."""
        current = self.head
        while current:
            print(current.data, end=" -> ")
            current = current.next
        print("None")

# --- Test Cases ---
print("--- Testing insert_at_end() ---")
linked_list = LinkedList()
linked_list.insert_at_end(10)
print("List after inserting 10:")
linked_list.traverse()

print("\nTest Case: Insert additional values")
linked_list.insert_at_end(20)
linked_list.insert_at_end(30)
print("List after inserting 20 and 30:")
linked_list.traverse()

print("\n--- Testing traverse() ---")
print("\nTest Case: Traverse an empty list")
empty_list = LinkedList()
```

```python
print("\n--- Testing traverse() ---")
print("\nTest Case: Traverse an empty list")
empty_list = LinkedList()
print("Traversing an empty list:")
empty_list.traverse()

print("\n--- Testing delete_value() ---")
print("\nTest Case: Delete from an empty list")
empty_list_for_delete = LinkedList()
print("Attempting to delete 5 from an empty list:")
empty_list_for_delete.delete_value(5)
print("List after attempted deletion:")
empty_list_for_delete.traverse()

print("\nTest Case: Delete the head node")
list_to_delete_head = LinkedList()
list_to_delete_head.insert_at_end(1)
list_to_delete_head.insert_at_end(2)
list_to_delete_head.insert_at_end(3)
print("\nOriginal list:")
list_to_delete_head.traverse()
print("Deleting head node (value 1):")
list_to_delete_head.delete_value(1)
print("List after deleting head:")
list_to_delete_head.traverse()

print("\nTest Case: Delete a node in the middle")
list_to_delete_middle = LinkedList()
list_to_delete_middle.insert_at_end(10)
list_to_delete_middle.insert_at_end(20)
list_to_delete_middle.insert_at_end(30)
list_to_delete_middle.insert_at_end(40)
print("\nOriginal list:")
list_to_delete_middle.traverse()
```

```python
list_to_delete_middle.traverse()
print("Deleting middle node (value 30):")
list_to_delete_middle.delete_value(30)
print("List after deleting middle node:")
list_to_delete_middle.traverse()

print("\nTest Case: Delete the last node")
list_to_delete_last = LinkedList()
list_to_delete_last.insert_at_end(100)
list_to_delete_last.insert_at_end(200)
list_to_delete_last.insert_at_end(300)
print("\nOriginal list:")
list_to_delete_last.traverse()
print("Deleting last node (value 300):")
list_to_delete_last.delete_value(300)
print("List after deleting last node:")
list_to_delete_last.traverse()

print("\nTest Case: Attempt to delete a value not in the list")
list_no_value = LinkedList()
list_no_value.insert_at_end(5)
list_no_value.insert_at_end(15)
list_no_value.insert_at_end(25)
print("\nOriginal list:")
list_no_value.traverse()
print("Attempting to delete value 100 (not in list):")
list_no_value.delete_value(100)
print("List after attempted deletion:")
list_no_value.traverse()

print("\nTest Case: Delete a value from a list with duplicate values")
list_with_duplicates = LinkedList()
list_with_duplicates.insert_at_end(5)
list_with_duplicates.insert_at_end(10)
```

```python
print("\nTest Case: Delete a value from a list with duplicate values")
list_with_duplicates = LinkedList()
list_with_duplicates.insert_at_end(5)
list_with_duplicates.insert_at_end(10)
list_with_duplicates.insert_at_end(5)
list_with_duplicates.insert_at_end(15)
list_with_duplicates.insert_at_end(5)
print("\nOriginal list:")
list_with_duplicates.traverse()
print("Deleting first occurrence of value 5:")
list_with_duplicates.delete_value(5)
print("List after deleting first occurrence of 5:")
list_with_duplicates.traverse()
```

```
--- Testing insert_at_end() ---
List after inserting 10:
10 -> None

Test Case: Insert additional values
List after inserting 20 and 30:
10 -> 20 -> 30 -> None

--- Testing traverse() ---

Test Case: Traverse an empty list
Traversing an empty list:
None

--- Testing delete_value() ---

Test Case: Delete from an empty list
Attempting to delete 5 from an empty list:
List is empty. Cannot delete.
List after attempted deletion:
None

Test Case: Delete the head node

Original list:
1 -> 2 -> 3 -> None
Deleting head node (value 1):
Deleted head node with value: 1
List after deleting head:
2 -> 3 -> None
```

```
Test Case: Delete a node in the middle

Original list:
10 -> 20 -> 30 -> 40 -> None
Deleting middle node (value 30):
List after deleting middle node:
10 -> 20 -> 40 -> None

Test Case: Delete the last node

Original list:
100 -> 200 -> 300 -> None
Deleting last node (value 300):
List after deleting last node:
100 -> 200 -> None

Test Case: Attempt to delete a value not in the list

Original list:
5 -> 15 -> 25 -> None
Attempting to delete value 100 (not in list):
Value 100 not found in the list.
List after attempted deletion:
5 -> 15 -> 25 -> None

Test Case: Delete a value from a list with duplicate values

Original list:
5 -> 10 -> 5 -> 15 -> 5 -> None
Deleting first occurrence of value 5:
Deleted head node with value: 5
List after deleting first occurrence of 5:
10 -> 5 -> 15 -> 5 -> None
```

Explanation:

Explanation

Certainly! The code you see implements a basic **Singly Linked List** data structure in Python. It's composed of two main classes: `Node` and `LinkedList`.

Here's a breakdown of the code and what each part does:

1. **`Node` Class (cell `cd05d182`):**
   - This class represents a single element (or node) within the linked list.
   - `__init__(self, data, next_node=None)`: The constructor for the `Node`.
     - `self.data`: Stores the actual value or data of the node.
     - `self.next`: Stores a reference (or "pointer") to the *next* node in the sequence. It's initialized to `None` by default, indicating the end of the list or sublist.
2. **`LinkedList` Class (cells `05f52146`, `bd75d654`, `0c01caa5`, `4fad55d8`):**
   - This class represents the entire linked list.
   - `__init__(self)`: The constructor for the `LinkedList`.
     - `self.head`: This is the most important attribute. It stores a reference to the *first* node in the list. It's initialized to `None` when the list is empty.
   - **`insert_at_end(self, data)` (cell `bd75d654`):**
     - This method adds a new node containing the given `data` to the *end* of the list.
     - It creates a `new_node`.
     - If the list is empty (`self.head is None`), the `new_node` becomes the `head`.
     - If the list is not empty, it traverses the list starting from the `head` until it reaches the last node (the one whose `next` is `None`).
     - Finally, it updates the `next` pointer of the last node to point to the `new_node`, effectively linking it to the end. The inline comments explain these pointer updates.
   - **`delete_value(self, value)` (cell `0c01caa5`):**
     - This method removes the *first* node it finds that contains the given `value`.
     - It handles several cases:
       - **Empty List:** If the list is empty, it prints a message and does nothing.
       - **Deleting the Head:** If the `head` node contains the `value`, it updates `self.head` to point to the *next* node, effectively removing the original head.
       - **Deleting Other Nodes:** It traverses the list, keeping track of the `current` node and the `previous` node. When it finds the node with the matching `value` (`current`), it updates the `previous` node's `next` pointer to point to `current.next`, bypassing and effectively removing `current`. Inline comments clarify these pointer changes.
       - **Value Not Found:** If the loop finishes without finding the `value`, it prints a message.
   - **`traverse(self)` (cell `4fad55d8`):**
     - This method iterates through the linked list from the `head` to the end.
     - It starts with a `current` pointer at the `head`.
     - It loops as long as `current` is not `None`.
     - In each iteration, it prints the `data` of the `current` node and moves `current` to the `next` node (`current = current.next`).
     - It prints " -> None" at the end to visually represent the list's structure and termination.

The subsequent cells (cell `dfa1bb64` and `99552937`) provide and implement various test cases to verify that the `insert_at_end`, `delete_value`, and `traverse` methods work correctly under different scenarios, including edge cases like empty lists, deleting the head, deleting the last node, and attempting to delete a non-existent value.

Overall, this code provides a fundamental implementation of a singly linked list, demonstrating how nodes are connected and how basic operations like adding, deleting, and iterating through elements are performed by manipulating the `next` pointers.

Task 4:

write a python program to Implement a Binary Search Tree with methods for insert(), search(), and inorder_traversal().Provide AI with a partially written Node and BST class.Ask AI to complete missing methods and add docstrings.Test with a list of integers and compare outputs of search() for present vs absent elements.

Code and Output:

Commands + Code + Text ▷ Run all

[7] 0s

```python
class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

class BST:
    def __init__(self):
        self.root = None

    def insert(self, data):
        if self.root is None:
            self.root = Node(data)
        else:
            self._insert_recursive(self.root, data)

    def _insert_recursive(self, current_node, data):
        if data < current_node.data:
            if current_node.left is None:
                current_node.left = Node(data)
            else:
                self._insert_recursive(current_node.left, data)
        elif data > current_node.data:
            if current_node.right is None:
                current_node.right = Node(data)
            else:
                self._insert_recursive(current_node.right, data)

    def search(self, data):
        return self._search_recursive(self.root, data)

    def _search_recursive(self, current_node, data):
        if current_node is None:
            return False
```

RAM Disk

Gemini

write a python program to Implement a Binary Search Tree with methods for insert(), search(), and inorder_traversal().Provide AI with a partially written Node and BST class.Ask AI to complete missing methods and add docstrings.Test with a list of integers and compare outputs of search() for present vs absent elements.

Happy to help! Here is a plan to implement a Binary Search Tree with insert, search, and inorder traversal methods, and test it with a list of integers.

- **Define node class**: Define the Node class for the Binary Search Tree.
- **Define bst class**: Define the Binary Search Tree class with methods for `insert()`, `search()`, and `inorder_traversal()`.
- **Implement bst methods**: Implement the `insert()`, `search()`, and `inorder_traversal()` methods within the BST class, ensuring they handle different scenarios correctly.
- **Add docstrings**: Add comprehensive docstrings to all classes and methods, explaining their purpose, parameters, and return values.
- **Test bst**: Create a list of integers, insert them into the BST, and test the `search()` method for both present and absent elements. Print the outputs.
- **Inorder traversal**: Perform an inorder traversal of the BST and print the elements to verify the tree structure.
- **Finish task**: Put the findings from the earlier stages into a format that anyone can read.

Generating...

Preparing

What can I help you build?

Gemini can make mistakes so double-check it and use code with caution. Learn more

```python
    def search(self, data):
        return self._search_recursive(self.root, data)

    def _search_recursive(self, current_node, data):
        if current_node is None:
            return False
        if current_node.data == data:
            return True
        elif data < current_node.data:
            return self._search_recursive(current_node.left, data)
        else:
            return self._search_recursive(current_node.right, data)

    def inorder_traversal(self):
        result = []
        self._inorder_recursive(self.root, result)
        return result

    def _inorder_recursive(self, current_node, result):
        if current_node:
            self._inorder_recursive(current_node.left, result)
            result.append(current_node.data)
            self._inorder_recursive(current_node.right, result)

# Test the implementation
int_list = [50, 30, 20, 40, 70, 60, 80]
bst = BST()
for number in int_list:
    bst.insert(number)

present_element = 40
search_result_present = bst.search(present_element)
print(f"Searching for {present_element}: {search_result_present}")
```

```python
absent_element = 90
search_result_absent = bst.search(absent_element)
print(f"Searching for {absent_element}: {search_result_absent}")

inorder_result = bst.inorder_traversal()
print("In-order traversal:", inorder_result)
```

```
Searching for 40: True
Searching for 90: False
In-order traversal: [20, 30, 40, 50, 60, 70, 80]
```

Explanation:

K explanation

Certainly! The code in cell `5b7a1f3a` implements a Binary Search Tree (BST) data structure in Python.

Here's a breakdown:

- `Node` **class:** This class represents a single node within the BST. Each node stores a `data` value and has pointers (`left` and `right`) to its left and right child nodes.
- `BST` **class:** This class represents the entire Binary Search Tree. It has a `root` attribute that points to the topmost node of the tree.
  - `__init__`: Initializes an empty BST by setting the `root` to `None`.
  - `insert(data)`: This method adds a new node with the given `data` to the BST while maintaining the BST properties (smaller values go to the left, larger values go to the right). It uses a recursive helper method `_insert_recursive` to find the correct position.
  - `search(data)`: This method checks if a node with the given `data` exists in the BST. It also uses a recursive helper method `_search_recursive` to efficiently traverse the tree.
  - `inorder_traversal()`: This method performs an in-order traversal of the BST, which visits the nodes in ascending order of their data values. It uses a recursive helper method `_inorder_recursive` to build a list of the visited nodes' data.

The code then creates a `BST` instance, inserts some integers into it, and demonstrates the `search` method for both a present and an absent element. Finally, it performs an in-order traversal and prints the sorted elements.

Task 5:

Write a python program to Implement a Graph using an adjacency list, with traversal methods BFS() and DFS().Start with an adjacency list dictionary.And to generate BFS and DFS implementations with inline comments.Compare recursive vs iterative DFS if suggested by AI.

Code and Output:

```python
import collections

class Graph:
    def __init__(self):
        self.adj_list = {}

    def bfs(self, start_node):
        queue = collections.deque([start_node])
        visited = {start_node}
        traversal_order = []

        while queue:
            current_node = queue.popleft()
            traversal_order.append(current_node)

            if current_node in self.adj_list:
                for neighbor in self.adj_list[current_node]:
                    if neighbor not in visited:
                        visited.add(neighbor)
                        queue.append(neighbor)
        return traversal_order

    def dfs_recursive(self, start_node, visited=None, traversal_order=None):
        if visited is None:
            visited = set()
        if traversal_order is None:
            traversal_order = []

        visited.add(start_node)
        traversal_order.append(start_node)

        if start_node in self.adj_list:
            for neighbor in self.adj_list[start_node]:
                if neighbor not in visited:
```

Gemini

Implement a Graph using an adjacency list, with traversal methods BFS() and DFS().Start with an adjacency list dictionary.And to generate BFS and DFS implementations with inline comments.Compare recursive vs iterative DFS if suggested by AI.

Okay, I can help you with that. Here's a plan to implement a Graph using an adjacency list and add BFS and DFS traversal methods. We can also look into comparing recursive and iterative DFS implementations.

- **Define graph class with adjacency list**: Define a `Graph` class that uses a dictionary to represent the adjacency list.
- **Implement bfs**: Implement the Breadth-First Search (BFS) traversal method for the graph.
- **Implement dfs (recursive)**: Implement the Depth-First Search (DFS) traversal method using a recursive approach.
- **Implement dfs (iterative)**: Implement the Depth-First Search (DFS) traversal method using an iterative approach.
- **Compare dfs implementations**: Discuss the differences and trade-offs between the recursive and iterative DFS implementations.
- **Test traversal methods**: Create a sample graph and test both BFS and DFS methods, printing the traversal results.
- **Add inline comments**: Add inline comments to the code to explain the steps in the BFS and DFS implementations.
- **Finish task**: Put the findings from the earlier stages into a format that anyone can read.

What can I help you build?

Gemini can make mistakes so double-check it and use code with caution. Learn more

```python
        if start_node in self.adj_list:
            for neighbor in self.adj_list[start_node]:
                if neighbor not in visited:
                    self.dfs_recursive(neighbor, visited, traversal_order)
        return traversal_order

    def dfs_iterative(self, start_node):
        traversal_order = []
        visited = set()
        stack = [start_node]
        visited.add(start_node)

        while stack:
            current_node = stack.pop()
            traversal_order.append(current_node)

            if current_node in self.adj_list:
                for neighbor in reversed(self.adj_list[current_node]):
                    if neighbor not in visited:
                        visited.add(neighbor)
                        stack.append(neighbor)
        return traversal_order

# Test the implementation
graph = Graph()
graph.adj_list = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}
```

```python
start_node = 'A'

bfs_result = graph.bfs(start_node)
print(f"BFS traversal starting from {start_node}: {bfs_result}")

dfs_recursive_result = graph.dfs_recursive(start_node)
print(f"Recursive DFS traversal starting from {start_node}: {dfs_recursive_result}")

dfs_iterative_result = graph.dfs_iterative(start_node)
print(f"Iterative DFS traversal starting from {start_node}: {dfs_iterative_result}")
```

```
BFS traversal starting from A: ['A', 'B', 'C', 'D', 'E', 'F']
Recursive DFS traversal starting from A: ['A', 'B', 'D', 'E', 'F', 'C']
Iterative DFS traversal starting from A: ['A', 'B', 'D', 'E', 'F', 'C']
```

Explanation:

explanation

Certainly! The code in cell `4c8cff06` implements a `Graph` data structure using an adjacency list and includes methods for Breadth-First Search (BFS) and Depth-First Search (DFS) traversals.

Here's a breakdown:

- **`Graph` class:** This class represents a graph. It uses a dictionary called `adj_list` to store the adjacency list representation. The keys of the dictionary are the nodes, and the values are lists of their neighboring nodes.
  - **`__init__`**: Initializes an empty graph with an empty adjacency list.
  - **`bfs(start_node)`**: Performs a Breadth-First Search starting from the `start_node`. It uses a queue to explore the graph level by level, ensuring that all nodes at the current depth are visited before moving to the next depth. A `visited` set is used to keep track of visited nodes and prevent infinite loops in graphs with cycles.
  - **`dfs_recursive(start_node, visited=None, traversal_order=None)`**: Performs a Depth-First Search starting from the `start_node` using a recursive approach. It explores as far as possible along each branch before backtracking. It uses a `visited` set to keep track of visited nodes and a `traversal_order` list to store the order of visited nodes.
  - **`dfs_iterative(start_node)`**: Performs a Depth-First Search starting from the `start_node` using an iterative approach with an explicit stack. This method simulates the recursive call stack to achieve the same traversal order. It also uses a `visited` set to track visited nodes. Note that the neighbors are pushed onto the stack in reverse order to match the typical recursive DFS traversal order when the adjacency list is processed in a specific order.

The code then creates a sample `Graph` instance with some nodes and edges defined in the `adj_list`. It then performs BFS, recursive DFS, and iterative DFS starting from node 'A' and prints the resulting traversal orders.