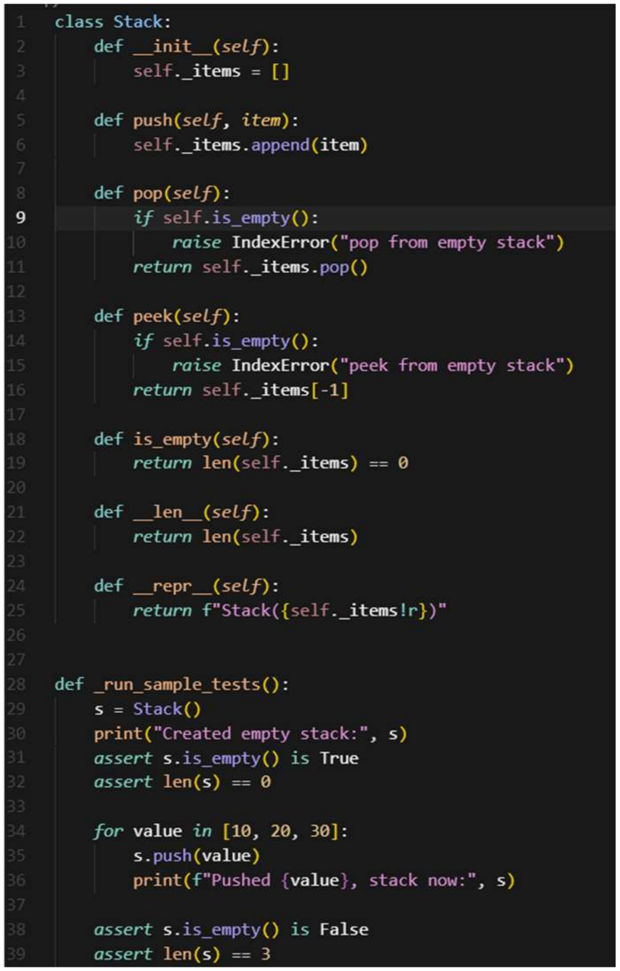
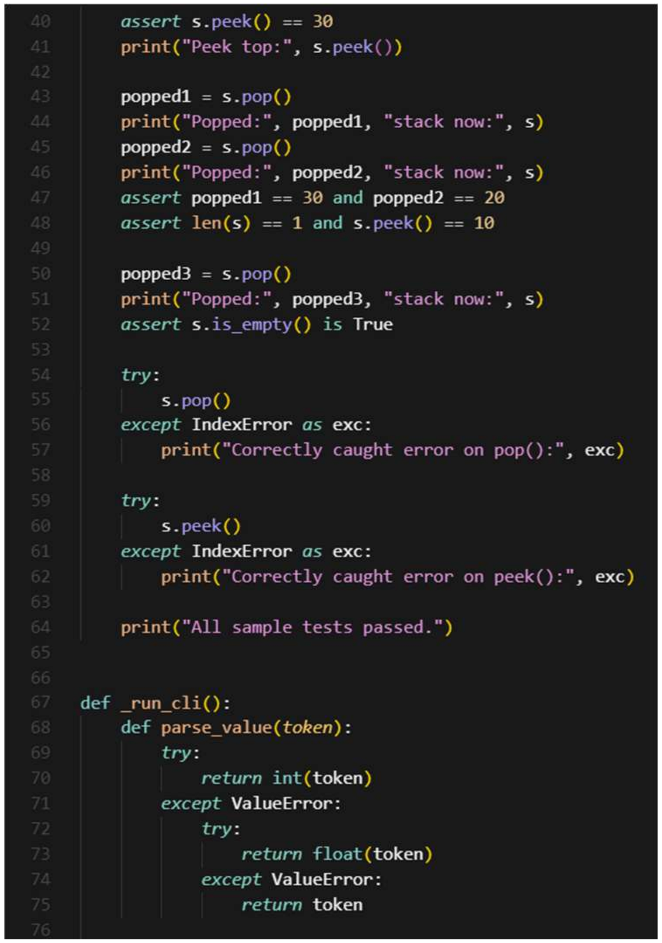
Task-1:

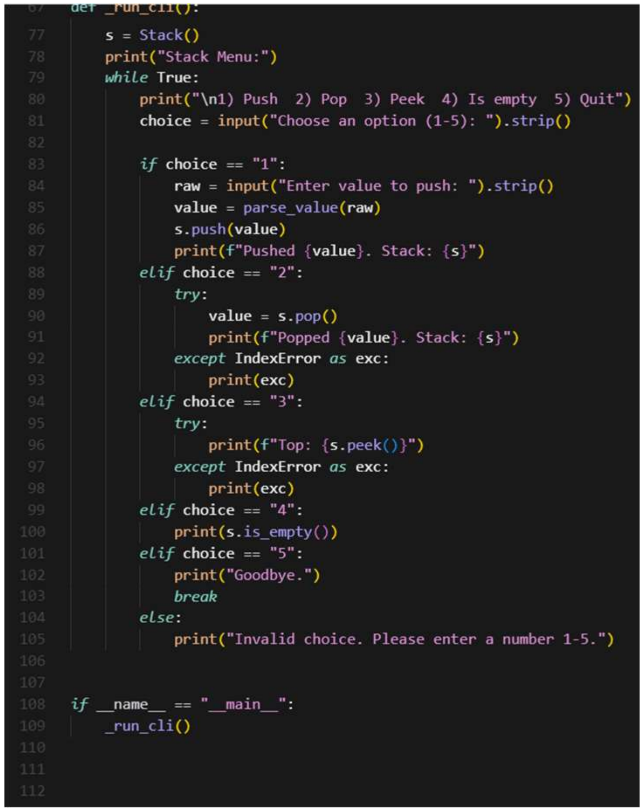
Prompt: write a python code by using ai to implement a **Stack** class only with following operations: push(), pop(), peek(), and is_empty().



```python
class Stack:
    def __init__(self):
        self._items = []

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

    def pop(self):
        if self.is_empty():
            raise IndexError("pop from empty stack")
        return self._items.pop()

    def peek(self):
        if self.is_empty():
            raise IndexError("peek from empty stack")
        return self._items[-1]

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

    def __len__(self):
        return len(self._items)

    def __repr__(self):
        return f"Stack({self._items!r})"


def _run_sample_tests():
    s = Stack()
    print("Created empty stack:", s)
    assert s.is_empty() is True
    assert len(s) == 0

    for value in [10, 20, 30]:
        s.push(value)
        print(f"Pushed {value}, stack now:", s)

    assert s.is_empty() is False
    assert len(s) == 3
```

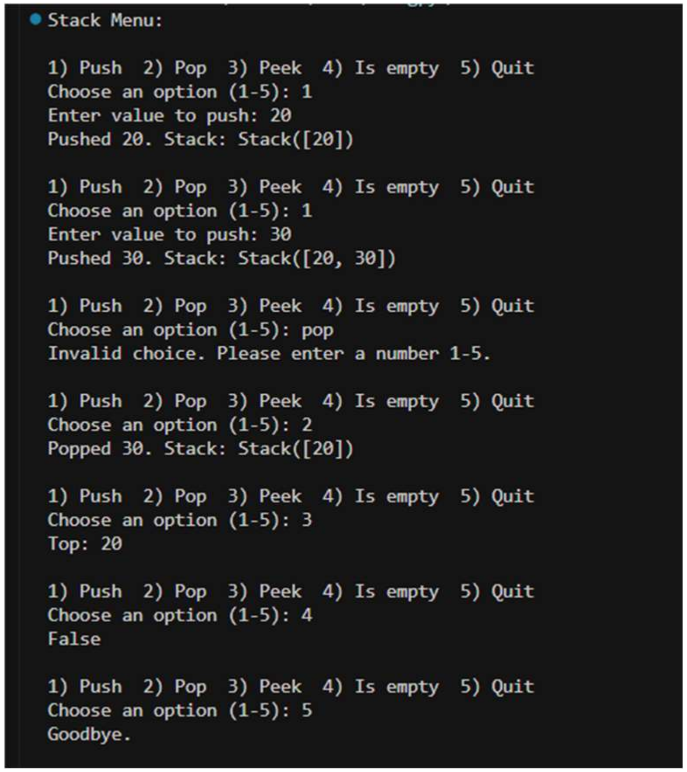


```python
    assert s.peek() == 30
    print("Peek top:", s.peek())

    popped1 = s.pop()
    print("Popped:", popped1, "stack now:", s)
    popped2 = s.pop()
    print("Popped:", popped2, "stack now:", s)
    assert popped1 == 30 and popped2 == 20
    assert len(s) == 1 and s.peek() == 10

    popped3 = s.pop()
    print("Popped:", popped3, "stack now:", s)
    assert s.is_empty() is True

    try:
        s.pop()
    except IndexError as exc:
        print("Correctly caught error on pop():", exc)

    try:
        s.peek()
    except IndexError as exc:
        print("Correctly caught error on peek():", exc)

    print("All sample tests passed.")


def _run_cli():
    def parse_value(token):
        try:
            return int(token)
        except ValueError:
            try:
                return float(token)
            except ValueError:
                return token

```



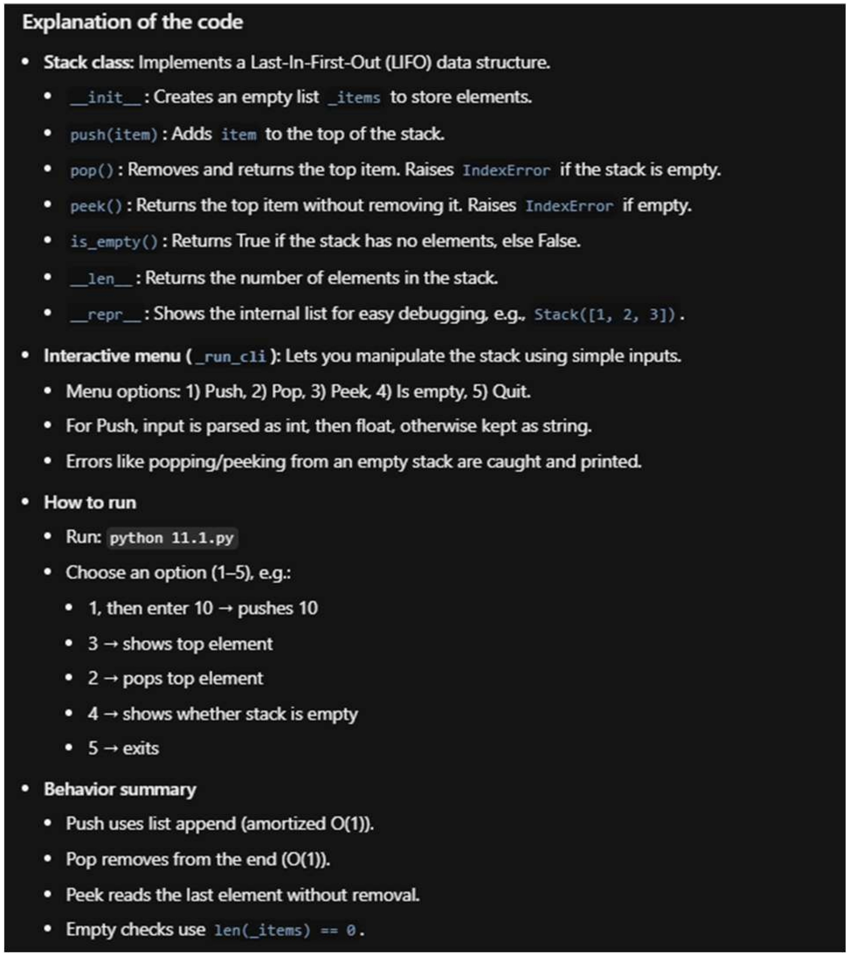
```python
def _run_cli():
    s = Stack()
    print("Stack Menu:")
    while True:
        print("\n1) Push  2) Pop  3) Peek  4) Is empty  5) Quit")
        choice = input("Choose an option (1-5): ").strip()

        if choice == "1":
            raw = input("Enter value to push: ").strip()
            value = parse_value(raw)
            s.push(value)
            print(f"Pushed {value}. Stack: {s}")
        elif choice == "2":
            try:
                value = s.pop()
                print(f"Popped {value}. Stack: {s}")
            except IndexError as exc:
                print(exc)
        elif choice == "3":
            try:
                print(f"Top: {s.peek()}")
            except IndexError as exc:
                print(exc)
        elif choice == "4":
            print(s.is_empty())
        elif choice == "5":
            print("Goodbye.")
            break
        else:
            print("Invalid choice. Please enter a number 1-5.")


if __name__ == "__main__":
    _run_cli()
```

Output:



```
Stack Menu:

1) Push  2) Pop  3) Peek  4) Is empty  5) Quit
Choose an option (1-5): 1
Enter value to push: 20
Pushed 20. Stack: Stack([20])

1) Push  2) Pop  3) Peek  4) Is empty  5) Quit
Choose an option (1-5): 1
Enter value to push: 30
Pushed 30. Stack: Stack([20, 30])

1) Push  2) Pop  3) Peek  4) Is empty  5) Quit
Choose an option (1-5): pop
Invalid choice. Please enter a number 1-5.

1) Push  2) Pop  3) Peek  4) Is empty  5) Quit
Choose an option (1-5): 2
Popped 30. Stack: Stack([20])

1) Push  2) Pop  3) Peek  4) Is empty  5) Quit
Choose an option (1-5): 3
Top: 20

1) Push  2) Pop  3) Peek  4) Is empty  5) Quit
Choose an option (1-5): 4
False

1) Push  2) Pop  3) Peek  4) Is empty  5) Quit
Choose an option (1-5): 5
Goodbye.
```

Explanation:

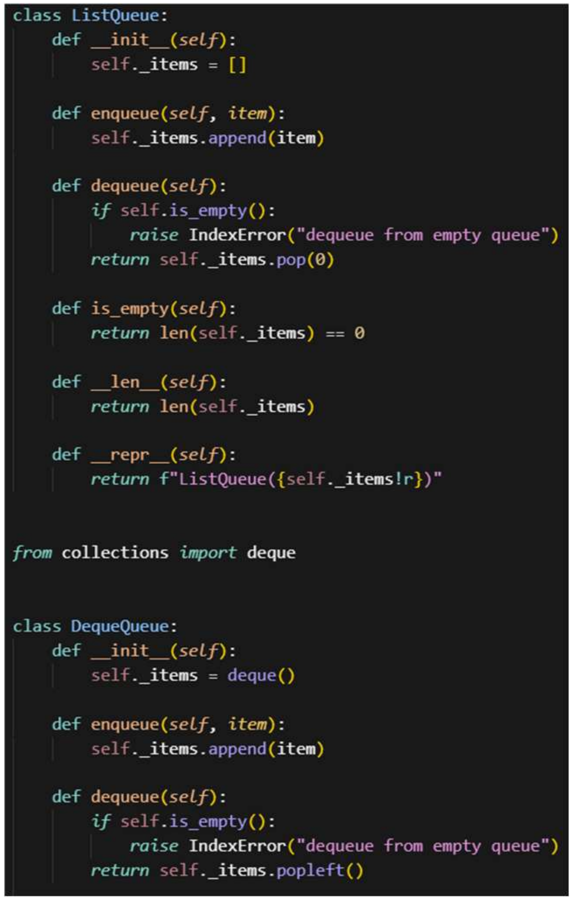


Explanation of the code

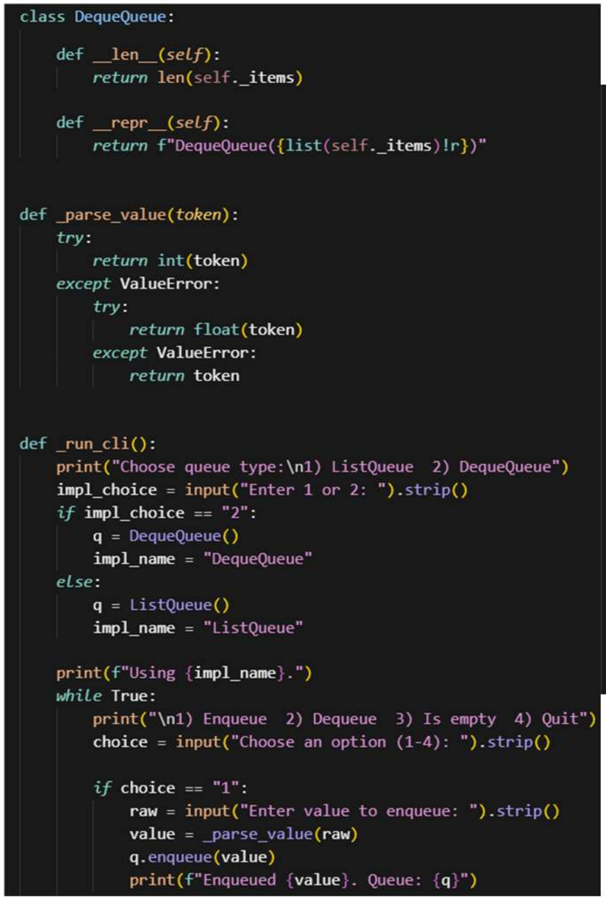
- **Stack class:** Implements a Last-In-First-Out (LIFO) data structure.
  - `__init__`: Creates an empty list `_items` to store elements.
  - `push(item)`: Adds `item` to the top of the stack.
  - `pop()`: Removes and returns the top item. Raises `IndexError` if the stack is empty.
  - `peek()`: Returns the top item without removing it. Raises `IndexError` if empty.
  - `is_empty()`: Returns True if the stack has no elements, else False.
  - `__len__`: Returns the number of elements in the stack.
  - `__repr__`: Shows the internal list for easy debugging, e.g., `Stack([1, 2, 3])`.
- **Interactive menu (`_run_cli`):** Lets you manipulate the stack using simple inputs.
  - Menu options: 1) Push, 2) Pop, 3) Peek, 4) Is empty, 5) Quit.
  - For Push, input is parsed as int, then float, otherwise kept as string.
  - Errors like popping/peeking from an empty stack are caught and printed.
- **How to run**
  - Run: `python 11.1.py`
  - Choose an option (1–5), e.g.:
    - 1, then enter 10 → pushes 10
    - 3 → shows top element
    - 2 → pops top element
    - 4 → shows whether stack is empty
    - 5 → exits
- **Behavior summary**
  - Push uses list append (amortized O(1)).
  - Pop removes from the end (O(1)).
  - Peek reads the last element without removal.
  - Empty checks use `len(_items) == 0`.

Task-2:

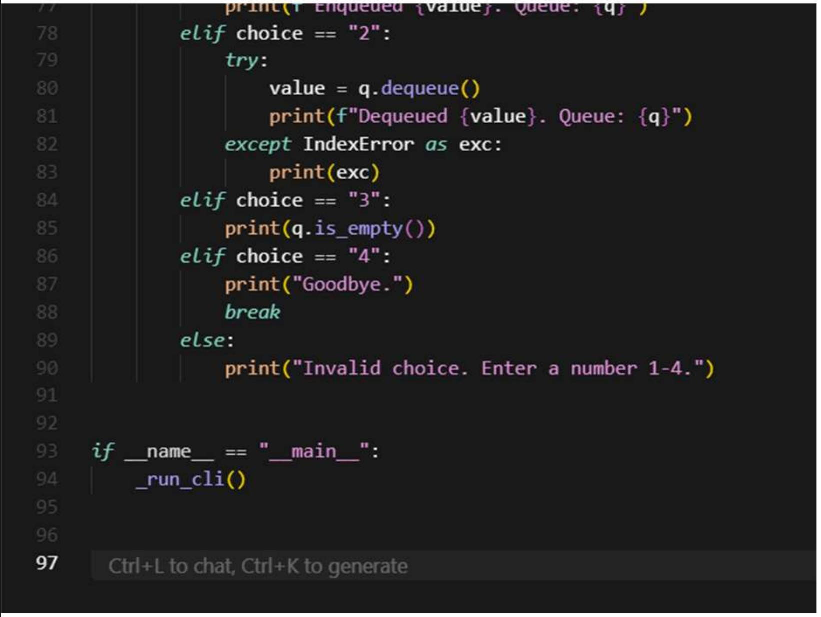
Prompt: write a python code to Implement a **Queue** with enqueue(), dequeue(), and is_empty() operations



```python
class ListQueue:
    def __init__(self):
        self._items = []

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

    def dequeue(self):
        if self.is_empty():
            raise IndexError("dequeue from empty queue")
        return self._items.pop(0)

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

    def __len__(self):
        return len(self._items)

    def __repr__(self):
        return f"ListQueue({self._items!r})"


from collections import deque


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

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

    def dequeue(self):
        if self.is_empty():
            raise IndexError("dequeue from empty queue")
        return self._items.popleft()
```

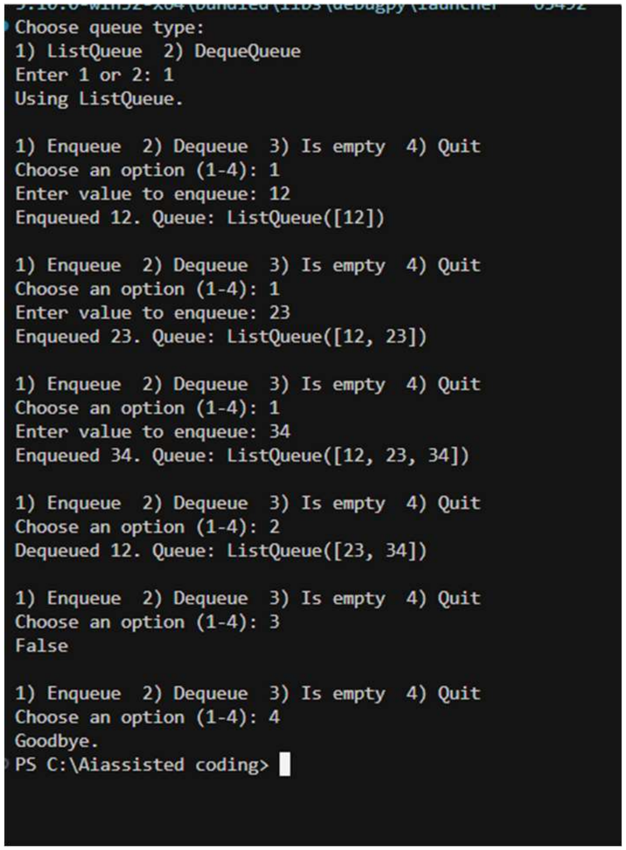


```python
class DequeQueue:

    def __len__(self):
        return len(self._items)

    def __repr__(self):
        return f"DequeQueue({list(self._items)!r})"


def _parse_value(token):
    try:
        return int(token)
    except ValueError:
        try:
            return float(token)
        except ValueError:
            return token


def _run_cli():
    print("Choose queue type:\n1) ListQueue  2) DequeQueue")
    impl_choice = input("Enter 1 or 2: ").strip()
    if impl_choice == "2":
        q = DequeQueue()
        impl_name = "DequeQueue"
    else:
        q = ListQueue()
        impl_name = "ListQueue"

    print(f"Using {impl_name}.")
    while True:
        print("\n1) Enqueue  2) Dequeue  3) Is empty  4) Quit")
        choice = input("Choose an option (1-4): ").strip()

        if choice == "1":
            raw = input("Enter value to enqueue: ").strip()
            value = _parse_value(raw)
            q.enqueue(value)
            print(f"Enqueued {value}. Queue: {q}")
```



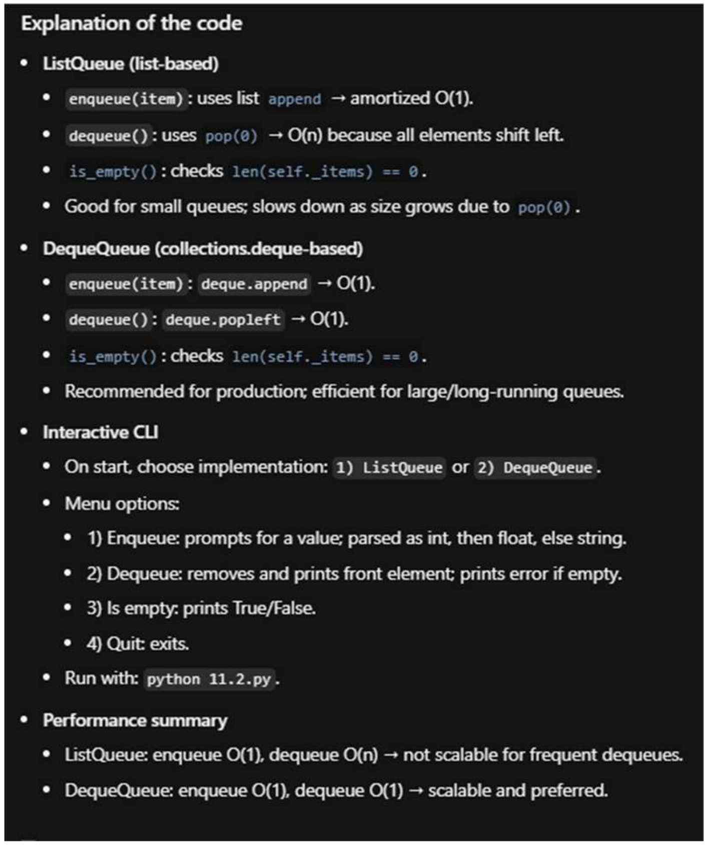
```python
            print(f"Enqueued {value}. Queue: {q}")
        elif choice == "2":
            try:
                value = q.dequeue()
                print(f"Dequeued {value}. Queue: {q}")
            except IndexError as exc:
                print(exc)
        elif choice == "3":
            print(q.is_empty())
        elif choice == "4":
            print("Goodbye.")
            break
        else:
            print("Invalid choice. Enter a number 1-4.")


if __name__ == "__main__":
    _run_cli()
```

Output:



```
Choose queue type:
1) ListQueue  2) DequeQueue
Enter 1 or 2: 1
Using ListQueue.

1) Enqueue  2) Dequeue  3) Is empty  4) Quit
Choose an option (1-4): 1
Enter value to enqueue: 12
Enqueued 12. Queue: ListQueue([12])

1) Enqueue  2) Dequeue  3) Is empty  4) Quit
Choose an option (1-4): 1
Enter value to enqueue: 23
Enqueued 23. Queue: ListQueue([12, 23])

1) Enqueue  2) Dequeue  3) Is empty  4) Quit
Choose an option (1-4): 1
Enter value to enqueue: 34
Enqueued 34. Queue: ListQueue([12, 23, 34])

1) Enqueue  2) Dequeue  3) Is empty  4) Quit
Choose an option (1-4): 2
Dequeued 12. Queue: ListQueue([23, 34])

1) Enqueue  2) Dequeue  3) Is empty  4) Quit
Choose an option (1-4): 3
False

1) Enqueue  2) Dequeue  3) Is empty  4) Quit
Choose an option (1-4): 4
Goodbye.
PS C:\Aiassisted coding>
```

Explanation:

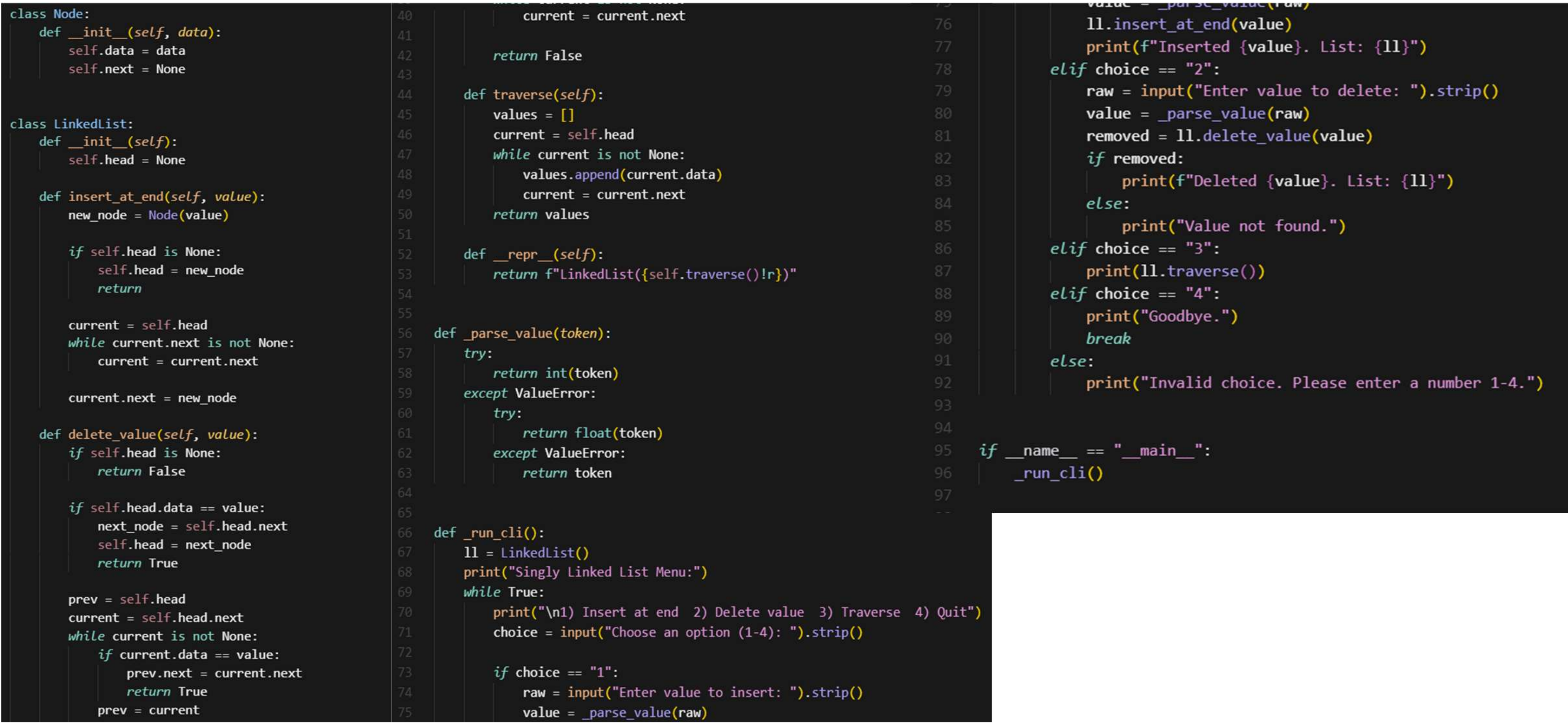


Explanation of the code

- **ListQueue (list-based)**
  - `enqueue(item)`: uses list `append` → amortized O(1).
  - `dequeue()`: uses `pop(0)` → O(n) because all elements shift left.
  - `is_empty()`: checks `len(self._items) == 0`.
  - Good for small queues; slows down as size grows due to `pop(0)`.
- **DequeQueue (collections.deque-based)**
  - `enqueue(item)`: `deque.append` → O(1).
  - `dequeue()`: `deque.popleft` → O(1).
  - `is_empty()`: checks `len(self._items) == 0`.
  - Recommended for production; efficient for large/long-running queues.
- **Interactive CLI**
  - On start, choose implementation: `1) ListQueue` or `2) DequeQueue`.
  - Menu options:
    - 1) Enqueue: prompts for a value; parsed as int, then float, else string.
    - 2) Dequeue: removes and prints front element; prints error if empty.
    - 3) Is empty: prints True/False.
    - 4) Quit: exits.
  - Run with: `python 11.2.py`.
- **Performance summary**
  - ListQueue: enqueue O(1), dequeue O(n) → not scalable for frequent dequeues.
  - DequeQueue: enqueue O(1), dequeue O(1) → scalable and preferred.

Task-3:

Prompt : write a python code to Implement a **Singly Linked List** with operations: insert_at_end(), delete_value(), and traverse().

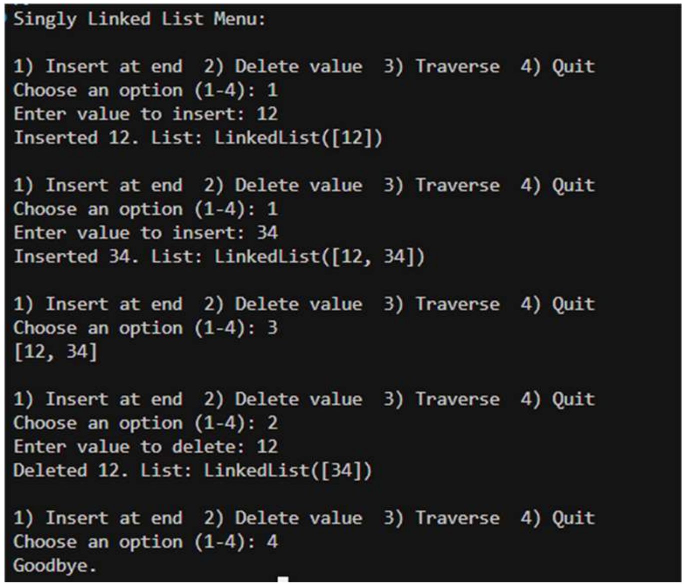


```python
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None


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

    def insert_at_end(self, value):
        new_node = Node(value)

        if self.head is None:
            self.head = new_node
            return

        current = self.head
        while current.next is not None:
            current = current.next

        current.next = new_node

    def delete_value(self, value):
        if self.head is None:
            return False

        if self.head.data == value:
            next_node = self.head.next
            self.head = next_node
            return True

        prev = self.head
        current = self.head.next
        while current is not None:
            if current.data == value:
                prev.next = current.next
                return True
            prev = current
```

```python
            current = current.next

        return False

    def traverse(self):
        values = []
        current = self.head
        while current is not None:
            values.append(current.data)
            current = current.next
        return values

    def __repr__(self):
        return f"LinkedList({self.traverse()!r})"


def _parse_value(token):
    try:
        return int(token)
    except ValueError:
        try:
            return float(token)
        except ValueError:
            return token


def _run_cli():
    ll = LinkedList()
    print("Singly Linked List Menu:")
    while True:
        print("\n1) Insert at end  2) Delete value  3) Traverse  4) Quit")
        choice = input("Choose an option (1-4): ").strip()

        if choice == "1":
            raw = input("Enter value to insert: ").strip()
            value = _parse_value(raw)
```

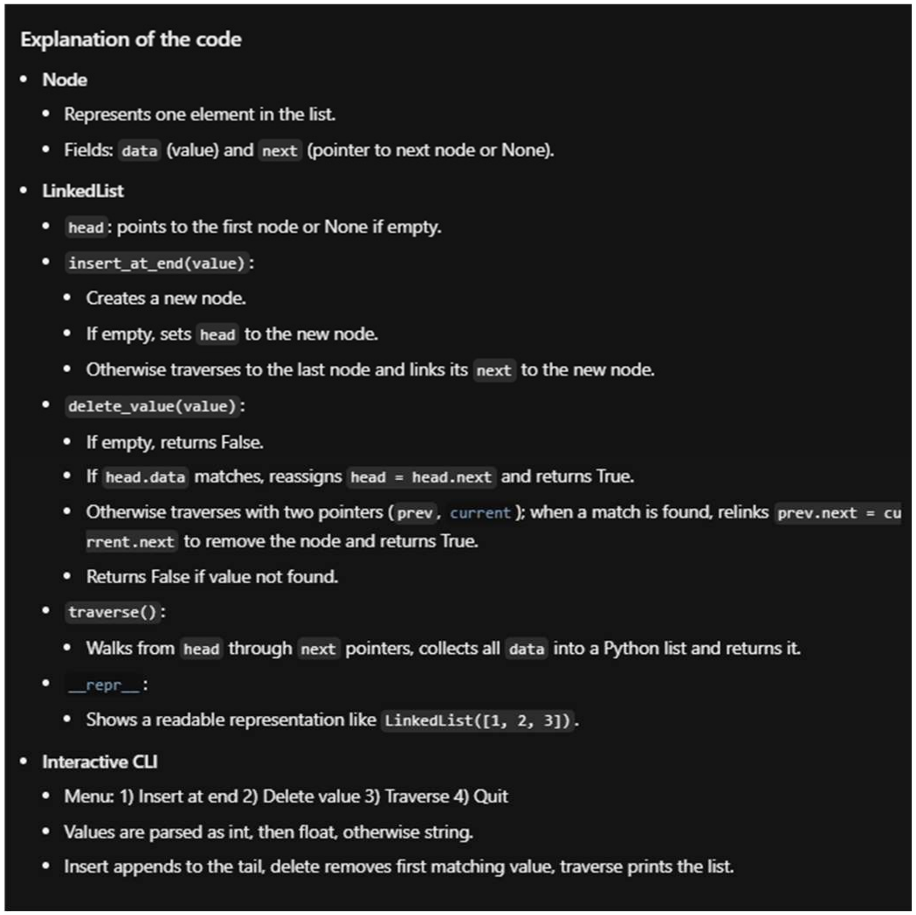
```python
            ll.insert_at_end(value)
            print(f"Inserted {value}. List: {ll}")
        elif choice == "2":
            raw = input("Enter value to delete: ").strip()
            value = _parse_value(raw)
            removed = ll.delete_value(value)
            if removed:
                print(f"Deleted {value}. List: {ll}")
            else:
                print("Value not found.")
        elif choice == "3":
            print(ll.traverse())
        elif choice == "4":
            print("Goodbye.")
            break
        else:
            print("Invalid choice. Please enter a number 1-4.")


if __name__ == "__main__":
    _run_cli()
```

Output:



```
Singly Linked List Menu:

1) Insert at end  2) Delete value  3) Traverse  4) Quit
Choose an option (1-4): 1
Enter value to insert: 12
Inserted 12. List: LinkedList([12])

1) Insert at end  2) Delete value  3) Traverse  4) Quit
Choose an option (1-4): 1
Enter value to insert: 34
Inserted 34. List: LinkedList([12, 34])

1) Insert at end  2) Delete value  3) Traverse  4) Quit
Choose an option (1-4): 3
[12, 34]

1) Insert at end  2) Delete value  3) Traverse  4) Quit
Choose an option (1-4): 2
Enter value to delete: 12
Deleted 12. List: LinkedList([34])

1) Insert at end  2) Delete value  3) Traverse  4) Quit
Choose an option (1-4): 4
Goodbye.
```

Explanation:

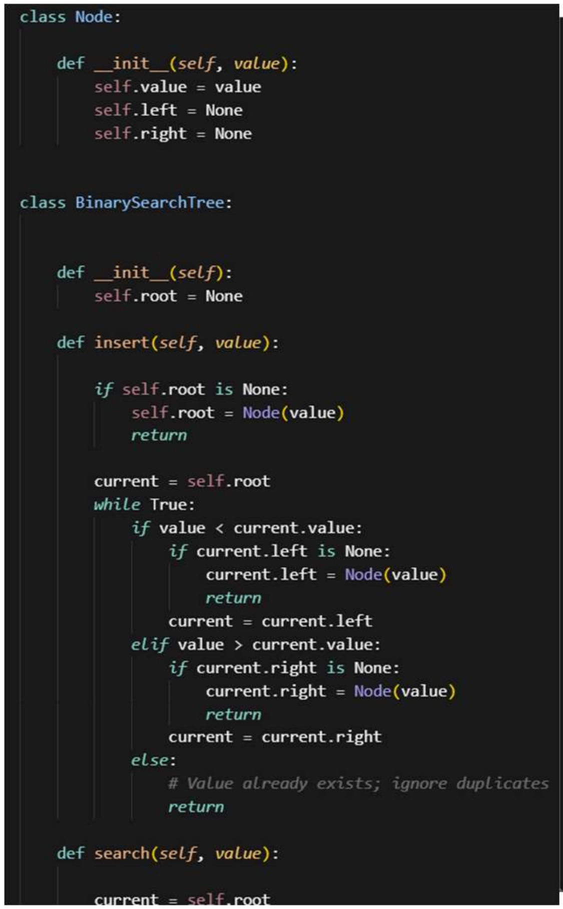


Explanation of the code

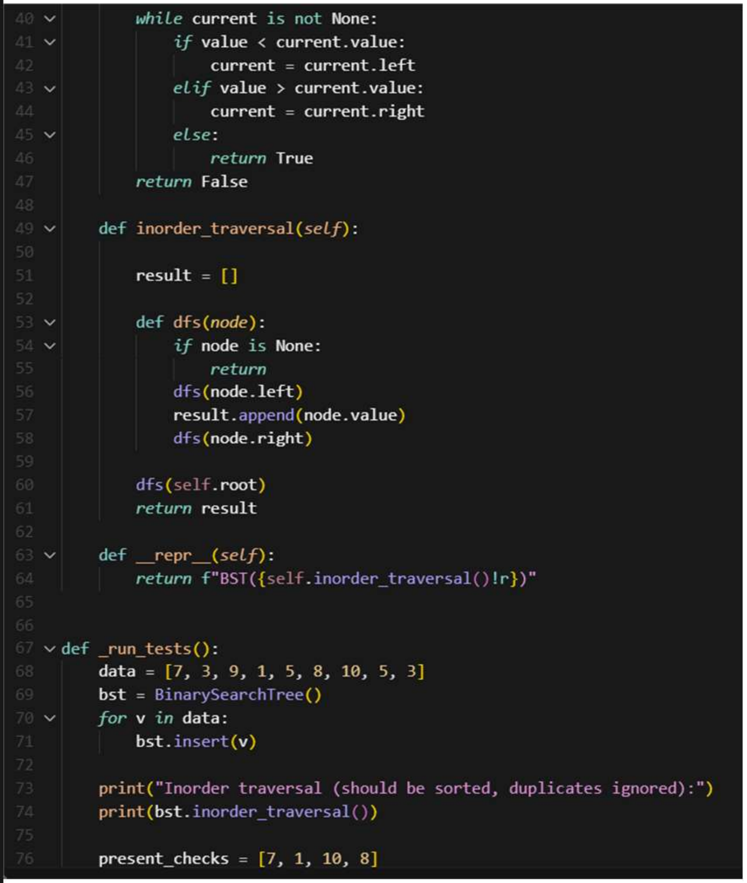
- **Node**
  - Represents one element in the list.
  - Fields: `data` (value) and `next` (pointer to next node or None).
- **LinkedList**
  - `head`: points to the first node or None if empty.
  - `insert_at_end(value)`:
    - Creates a new node.
    - If empty, sets `head` to the new node.
    - Otherwise traverses to the last node and links its `next` to the new node.
  - `delete_value(value)`:
    - If empty, returns False.
    - If `head.data` matches, reassigns `head = head.next` and returns True.
    - Otherwise traverses with two pointers (`prev`, `current`); when a match is found, relinks `prev.next = current.next` to remove the node and returns True.
    - Returns False if value not found.
  - `traverse()`:
    - Walks from `head` through `next` pointers, collects all `data` into a Python list and returns it.
  - `__repr__`:
    - Shows a readable representation like `LinkedList([1, 2, 3])`.
- **Interactive CLI**
  - Menu: 1) Insert at end 2) Delete value 3) Traverse 4) Quit
  - Values are parsed as int, then float, otherwise string.
  - Insert appends to the tail, delete removes first matching value, traverse prints the list.

Task-4:

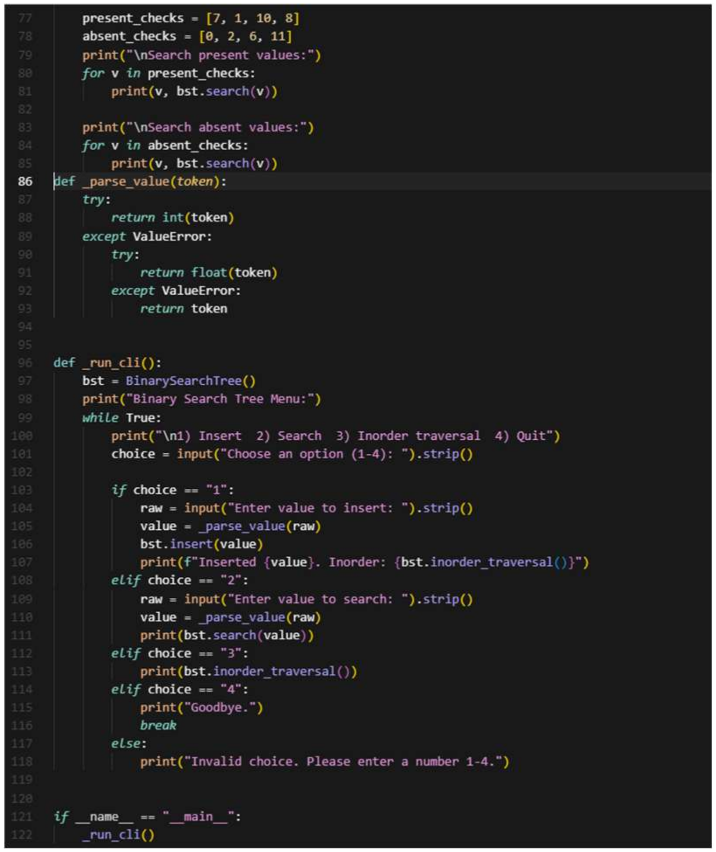
Prompt: write a python code to Implement a **Binary Search Tree** with methods for insert(), search(), and inorder_traversal().



```python
class Node:

    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None


class BinarySearchTree:


    def __init__(self):
        self.root = None

    def insert(self, value):

        if self.root is None:
            self.root = Node(value)
            return

        current = self.root
        while True:
            if value < current.value:
                if current.left is None:
                    current.left = Node(value)
                    return
                current = current.left
            elif value > current.value:
                if current.right is None:
                    current.right = Node(value)
                    return
                current = current.right
            else:
                # Value already exists; ignore duplicates
                return

    def search(self, value):

        current = self.root
```

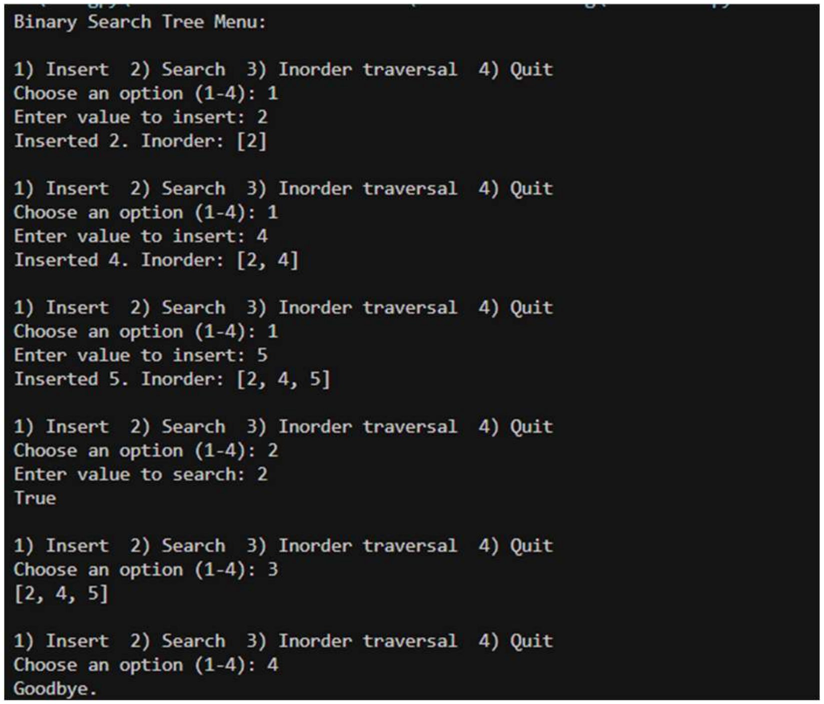


```python
        while current is not None:
            if value < current.value:
                current = current.left
            elif value > current.value:
                current = current.right
            else:
                return True
        return False

    def inorder_traversal(self):

        result = []

        def dfs(node):
            if node is None:
                return
            dfs(node.left)
            result.append(node.value)
            dfs(node.right)

        dfs(self.root)
        return result

    def __repr__(self):
        return f"BST({self.inorder_traversal()!r})"


def _run_tests():
    data = [7, 3, 9, 1, 5, 8, 10, 5, 3]
    bst = BinarySearchTree()
    for v in data:
        bst.insert(v)

    print("Inorder traversal (should be sorted, duplicates ignored):")
    print(bst.inorder_traversal())

    present_checks = [7, 1, 10, 8]
```



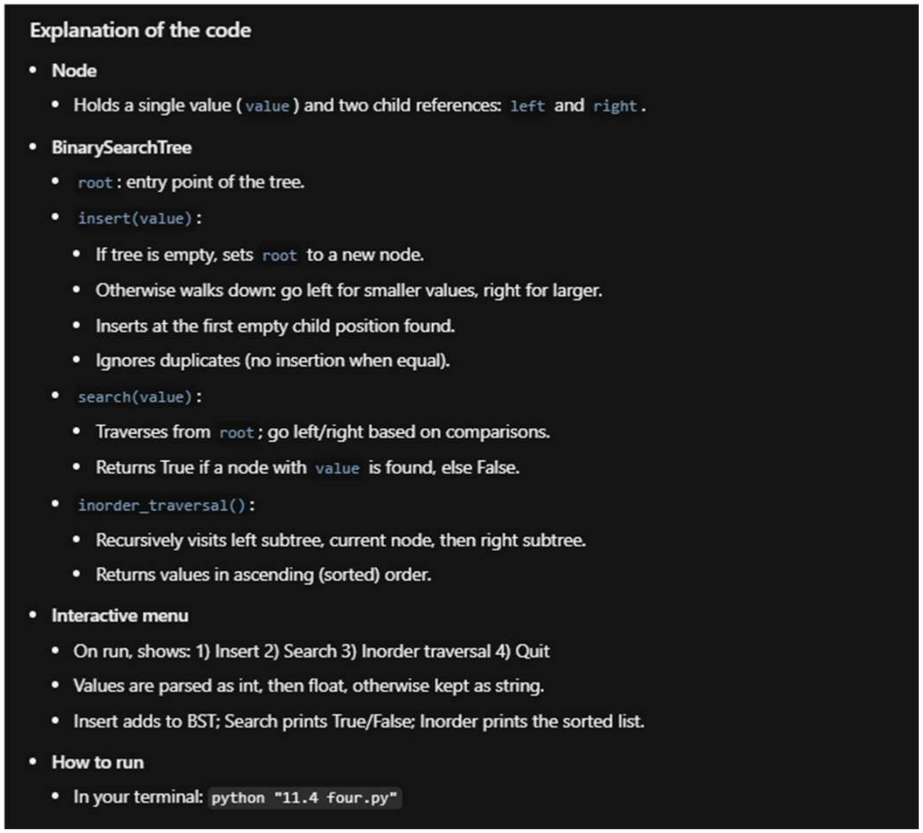
```python
    present_checks = [7, 1, 10, 8]
    absent_checks = [0, 2, 6, 11]
    print("\nSearch present values:")
    for v in present_checks:
        print(v, bst.search(v))

    print("\nSearch absent values:")
    for v in absent_checks:
        print(v, bst.search(v))
def _parse_value(token):
    try:
        return int(token)
    except ValueError:
        try:
            return float(token)
        except ValueError:
            return token


def _run_cli():
    bst = BinarySearchTree()
    print("Binary Search Tree Menu:")
    while True:
        print("\n1) Insert  2) Search  3) Inorder traversal  4) Quit")
        choice = input("Choose an option (1-4): ").strip()

        if choice == "1":
            raw = input("Enter value to insert: ").strip()
            value = _parse_value(raw)
            bst.insert(value)
            print(f"Inserted {value}. Inorder: {bst.inorder_traversal()}")
        elif choice == "2":
            raw = input("Enter value to search: ").strip()
            value = _parse_value(raw)
            print(bst.search(value))
        elif choice == "3":
            print(bst.inorder_traversal())
        elif choice == "4":
            print("Goodbye.")
            break
        else:
            print("Invalid choice. Please enter a number 1-4.")


if __name__ == "__main__":
    _run_cli()
```

Output:



```
Binary Search Tree Menu:

1) Insert  2) Search  3) Inorder traversal  4) Quit
Choose an option (1-4): 1
Enter value to insert: 2
Inserted 2. Inorder: [2]

1) Insert  2) Search  3) Inorder traversal  4) Quit
Choose an option (1-4): 1
Enter value to insert: 4
Inserted 4. Inorder: [2, 4]

1) Insert  2) Search  3) Inorder traversal  4) Quit
Choose an option (1-4): 1
Enter value to insert: 5
Inserted 5. Inorder: [2, 4, 5]

1) Insert  2) Search  3) Inorder traversal  4) Quit
Choose an option (1-4): 2
Enter value to search: 2
True

1) Insert  2) Search  3) Inorder traversal  4) Quit
Choose an option (1-4): 3
[2, 4, 5]

1) Insert  2) Search  3) Inorder traversal  4) Quit
Choose an option (1-4): 4
Goodbye.
```

Explanation:

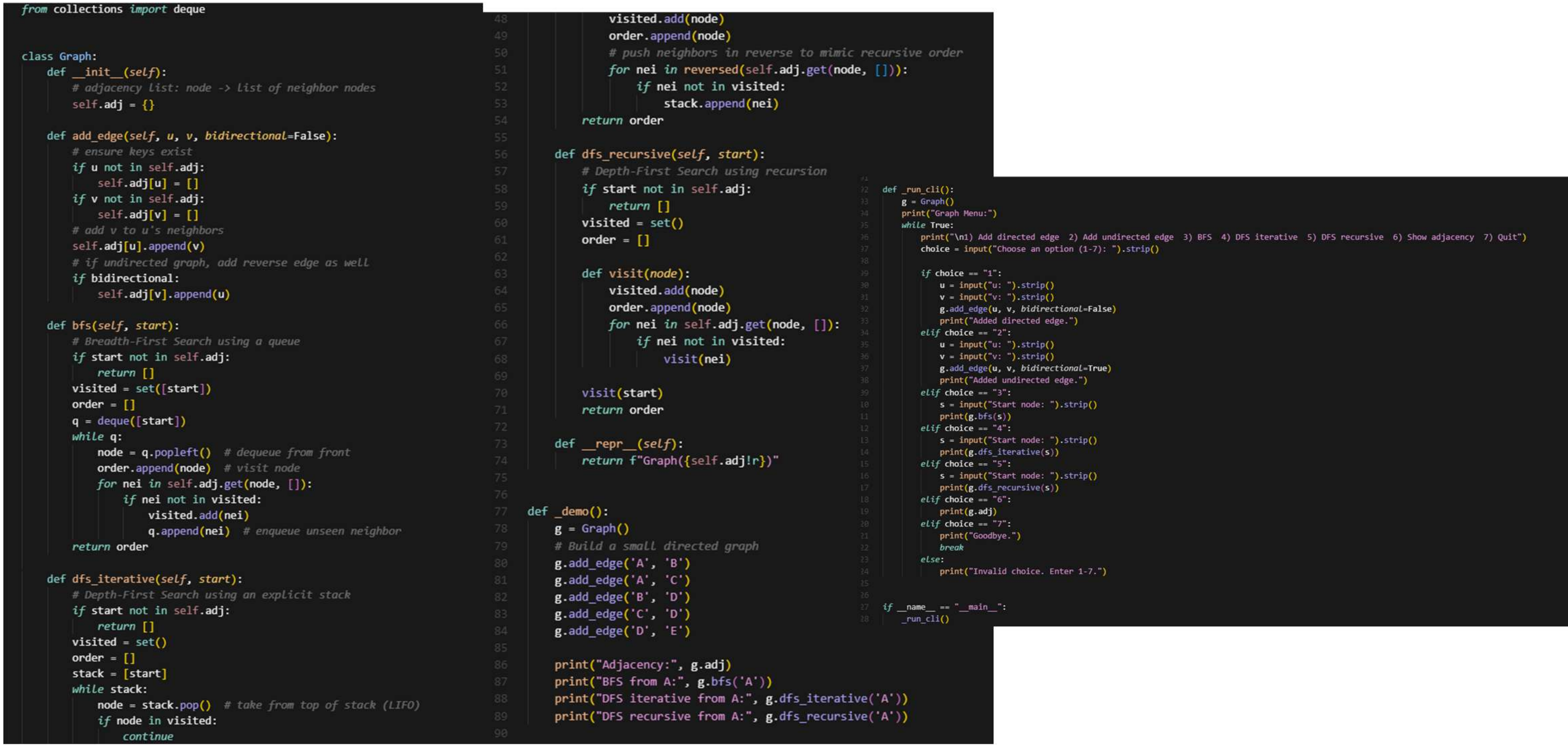


Explanation of the code

- **Node**
  - Holds a single value (`value`) and two child references: `left` and `right`.
- **BinarySearchTree**
  - `root`: entry point of the tree.
  - `insert(value)`:
    - If tree is empty, sets `root` to a new node.
    - Otherwise walks down: go left for smaller values, right for larger.
    - Inserts at the first empty child position found.
    - Ignores duplicates (no insertion when equal).
  - `search(value)`:
    - Traverses from `root`; go left/right based on comparisons.
    - Returns True if a node with `value` is found, else False.
  - `inorder_traversal()`:
    - Recursively visits left subtree, current node, then right subtree.
    - Returns values in ascending (sorted) order.
- **Interactive menu**
  - On run, shows: 1) Insert 2) Search 3) Inorder traversal 4) Quit
  - Values are parsed as int, then float, otherwise kept as string.
  - Insert adds to BST; Search prints True/False; Inorder prints the sorted list.
- **How to run**
  - In your terminal: `python "11.4 four.py"`

Task-5:

Prompt: give a python code to Implement a **Graph** using an adjacency list, with traversal methods BFS() and DFS().



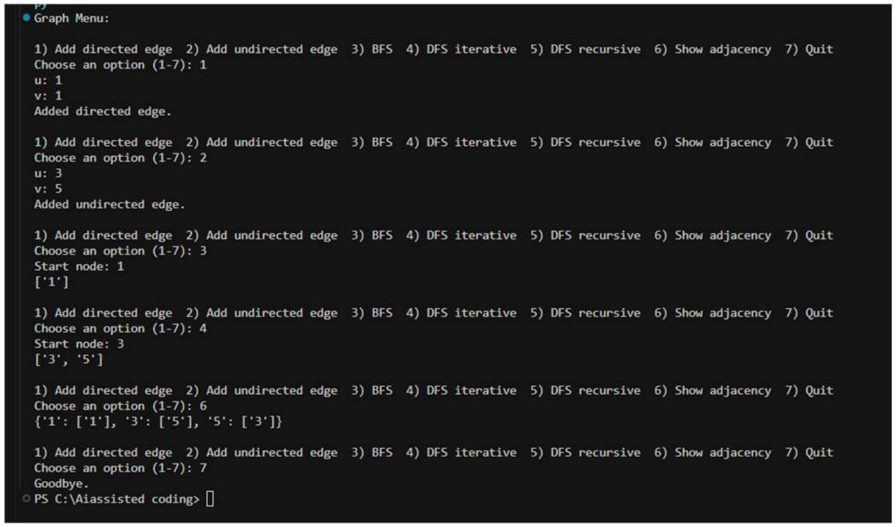
```python
from collections import deque


class Graph:
    def __init__(self):
        # adjacency list: node -> list of neighbor nodes
        self.adj = {}

    def add_edge(self, u, v, bidirectional=False):
        # ensure keys exist
        if u not in self.adj:
            self.adj[u] = []
        if v not in self.adj:
            self.adj[v] = []
        # add v to u's neighbors
        self.adj[u].append(v)
        # if undirected graph, add reverse edge as well
        if bidirectional:
            self.adj[v].append(u)

    def bfs(self, start):
        # Breadth-First Search using a queue
        if start not in self.adj:
            return []
        visited = set([start])
        order = []
        q = deque([start])
        while q:
            node = q.popleft()  # dequeue from front
            order.append(node)  # visit node
            for nei in self.adj.get(node, []):
                if nei not in visited:
                    visited.add(nei)
                    q.append(nei)  # enqueue unseen neighbor
        return order

    def dfs_iterative(self, start):
        # Depth-First Search using an explicit stack
        if start not in self.adj:
            return []
        visited = set()
        order = []
        stack = [start]
        while stack:
            node = stack.pop()  # take from top of stack (LIFO)
            if node in visited:
                continue
```

```python
            visited.add(node)
            order.append(node)
            # push neighbors in reverse to mimic recursive order
            for nei in reversed(self.adj.get(node, [])):
                if nei not in visited:
                    stack.append(nei)
        return order

    def dfs_recursive(self, start):
        # Depth-First Search using recursion
        if start not in self.adj:
            return []
        visited = set()
        order = []

        def visit(node):
            visited.add(node)
            order.append(node)
            for nei in self.adj.get(node, []):
                if nei not in visited:
                    visit(nei)

        visit(start)
        return order

    def __repr__(self):
        return f"Graph({self.adj!r})"


def _demo():
    g = Graph()
    # Build a small directed graph
    g.add_edge('A', 'B')
    g.add_edge('A', 'C')
    g.add_edge('B', 'D')
    g.add_edge('C', 'D')
    g.add_edge('D', 'E')

    print("Adjacency:", g.adj)
    print("BFS from A:", g.bfs('A'))
    print("DFS iterative from A:", g.dfs_iterative('A'))
    print("DFS recursive from A:", g.dfs_recursive('A'))
```

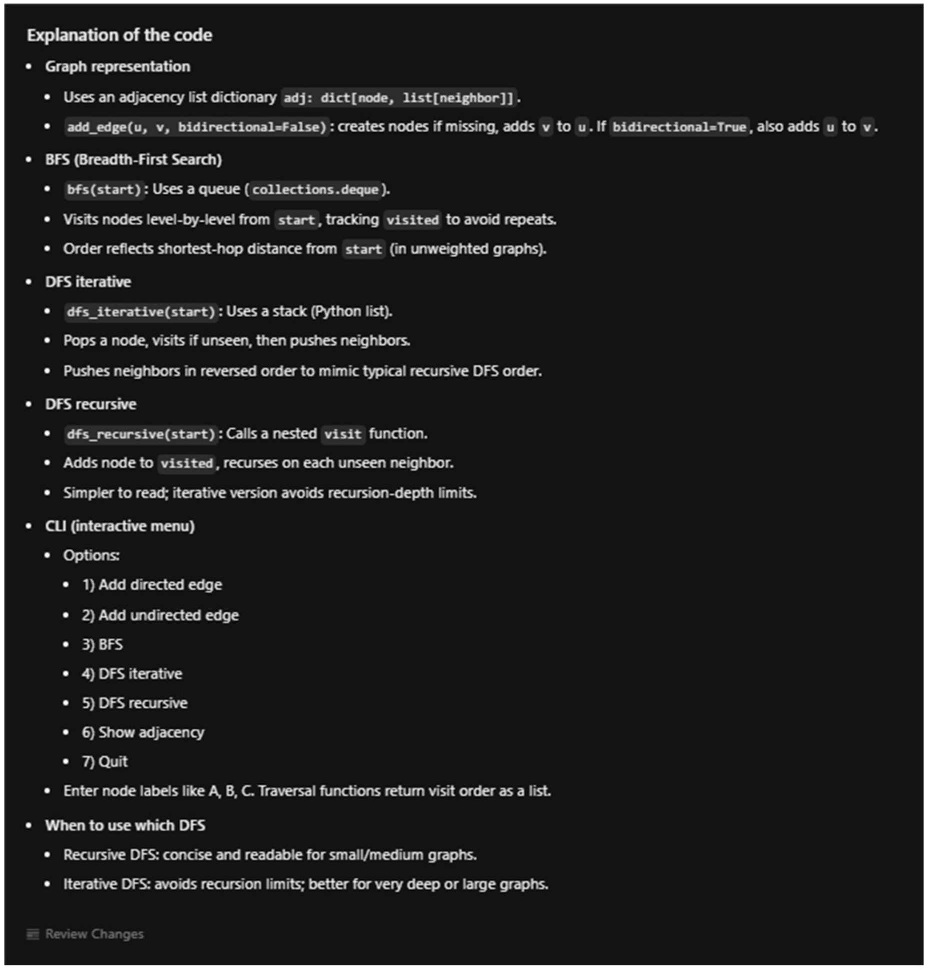
```python
def _run_cli():
    g = Graph()
    print("Graph Menu:")
    while True:
        print("\n1) Add directed edge  2) Add undirected edge  3) BFS  4) DFS iterative  5) DFS recursive  6) Show adjacency  7) Quit")
        choice = input("Choose an option (1-7): ").strip()

        if choice == "1":
            u = input("u: ").strip()
            v = input("v: ").strip()
            g.add_edge(u, v, bidirectional=False)
            print("Added directed edge.")
        elif choice == "2":
            u = input("u: ").strip()
            v = input("v: ").strip()
            g.add_edge(u, v, bidirectional=True)
            print("Added undirected edge.")
        elif choice == "3":
            s = input("Start node: ").strip()
            print(g.bfs(s))
        elif choice == "4":
            s = input("Start node: ").strip()
            print(g.dfs_iterative(s))
        elif choice == "5":
            s = input("Start node: ").strip()
            print(g.dfs_recursive(s))
        elif choice == "6":
            print(g.adj)
        elif choice == "7":
            print("Goodbye.")
            break
        else:
            print("Invalid choice. Enter 1-7.")


if __name__ == "__main__":
    _run_cli()
```

Explanation:

Output:



```
Graph Menu:

1) Add directed edge  2) Add undirected edge  3) BFS  4) DFS iterative  5) DFS recursive  6) Show adjacency  7) Quit
Choose an option (1-7): 1
u: 1
v: 1
Added directed edge.

1) Add directed edge  2) Add undirected edge  3) BFS  4) DFS iterative  5) DFS recursive  6) Show adjacency  7) Quit
Choose an option (1-7): 2
u: 3
v: 5
Added undirected edge.

1) Add directed edge  2) Add undirected edge  3) BFS  4) DFS iterative  5) DFS recursive  6) Show adjacency  7) Quit
Choose an option (1-7): 3
Start node: 1
['1']

1) Add directed edge  2) Add undirected edge  3) BFS  4) DFS iterative  5) DFS recursive  6) Show adjacency  7) Quit
Choose an option (1-7): 4
Start node: 3
['3', '5']

1) Add directed edge  2) Add undirected edge  3) BFS  4) DFS iterative  5) DFS recursive  6) Show adjacency  7) Quit
Choose an option (1-7): 6
{'1': ['1'], '3': ['5'], '5': ['3']}

1) Add directed edge  2) Add undirected edge  3) BFS  4) DFS iterative  5) DFS recursive  6) Show adjacency  7) Quit
Choose an option (1-7): 7
Goodbye.
PS C:\Aiassisted coding> 
```



Explanation of the code

- **Graph representation**
  - Uses an adjacency list dictionary `adj: dict[node, list[neighbor]]`.
  - `add_edge(u, v, bidirectional=False)`: creates nodes if missing, adds `v` to `u`. If `bidirectional=True`, also adds `u` to `v`.
- **BFS (Breadth-First Search)**
  - `bfs(start)`: Uses a queue (`collections.deque`).
  - Visits nodes level-by-level from `start`, tracking `visited` to avoid repeats.
  - Order reflects shortest-hop distance from `start` (in unweighted graphs).
- **DFS iterative**
  - `dfs_iterative(start)`: Uses a stack (Python list).
  - Pops a node, visits if unseen, then pushes neighbors.
  - Pushes neighbors in reversed order to mimic typical recursive DFS order.
- **DFS recursive**
  - `dfs_recursive(start)`: Calls a nested `visit` function.
  - Adds node to `visited`, recurses on each unseen neighbor.
  - Simpler to read; iterative version avoids recursion-depth limits.
- **CLI (interactive menu)**
  - Options:
    - 1) Add directed edge
    - 2) Add undirected edge
    - 3) BFS
    - 4) DFS iterative
    - 5) DFS recursive
    - 6) Show adjacency
    - 7) Quit
  - Enter node labels like A, B, C. Traversal functions return visit order as a list.
- **When to use which DFS**
  - Recursive DFS: concise and readable for small/medium graphs.
  - Iterative DFS: avoids recursion limits; better for very deep or large graphs.