CODE:

```
import tkinter as tk
from tkinter import messagebox
import math

class GraphGUI:
    def __init__(self, root):
        self.root = root
        self.root.title("Graph Editor - BFS Traversal")

        self.canvas = tk.Canvas(root, width=800, height=600,
bg="white")
        self.canvas.pack()

        self.nodes = {}  # Stores node data: {node_id: {"x": x, "y":
y, "oval": oval_obj, "text": text_obj}}
        self.edges = []  # Stores edge data: [{"nodes": (node1,
node2), "weight": w, "line": line_obj, "label": label_obj, "rect":
rect_obj}]
        self.adjacency = {} # Stores adjacency list for graph
traversals: {node_id: [neighbor1, neighbor2, ...]}
        self.node_id = 0
        self.selected_nodes_for_edge = [] # Used for selecting two
nodes to create an edge

        # To store traversal order
        self.bfs_order = []

        # Bindings for mouse events
        self.canvas.bind("<Button-1>", self.mouse_click)

        # Button frame for actions
        button_frame = tk.Frame(root)
        button_frame.pack(pady=5)

        tk.Button(button_frame, text="Run BFS",
command=self.run_bfs).pack(side=tk.LEFT, padx=5)
        tk.Button(button_frame, text="Delete Node",
command=self.delete_node_mode).pack(side=tk.LEFT, padx=5)
        tk.Button(button_frame, text="Delete Edge",
command=self.delete_edge_mode).pack(side=tk.LEFT, padx=5)
        tk.Button(button_frame, text="Clear Graph",
command=self.clear_graph).pack(side=tk.LEFT, padx=5)

        # Labels for displaying traversal order
        self.bfs_label = tk.Label(root, text="BFS Traversal: ",
font=("Arial", 12))
        self.bfs_label.pack(pady=2)

        self.current_mode_label = tk.Label(root, text="Mode: Add
Node/Connect Nodes", font=("Arial", 10), fg="blue")
        self.current_mode_label.pack(pady=2)

    def mouse_click(self, event):
        self.reset_canvas_bindings() # Ensure default bindings are
active
        clicked_node = self.get_node_at(event.x, event.y)

        if clicked_node is None:
            # If clicked on empty space, add a new node
            self.add_node(event.x, event.y)
            self.clear_selection_for_edge() # Clear any pending edge
selection
        else:
            # If clicked on an existing node, handle potential edge
creation
            self._handle_node_click_for_edge(clicked_node)

    def _handle_node_click_for_edge(self, clicked_node):
        if clicked_node not in self.selected_nodes_for_edge:
            self.selected_nodes_for_edge.append(clicked_node)
            self.highlight_node(clicked_node, "orange") # Highlight
selected nodes
        else:
            # If the same node is clicked again, deselect it
            self.selected_nodes_for_edge.remove(clicked_node)
            self.highlight_node(clicked_node, "lightblue") # Reset
```

```
color

        if len(self.selected_nodes_for_edge) == 2:
            node1, node2 = self.selected_nodes_for_edge
            if node1 == node2:
                messagebox.showwarning("Invalid Edge", "Cannot create
an edge to the same node.")
            else:
                self.add_edge(node1, node2, 1) # Default weight 1

            self.clear_selection_for_edge() # Reset selection after
attempting edge creation


    def add_node(self, x, y):
        r = 20
        node_name = str(self.node_id)
        oval = self.canvas.create_oval(x-r, y-r, x+r, y+r,
fill="lightblue", outline="black", width=2)
        text = self.canvas.create_text(x, y, text=node_name,
font=("Arial", 10, "bold"))

        self.nodes[node_name] = {
            "x": x, "y": y,
            "oval": oval,
            "text": text
        }
        self.adjacency[node_name] = []
        self.node_id += 1

    def get_node_at(self, x, y):
        for node, data in self.nodes.items():
            dx = x - data["x"]
            dy = y - data["y"]
            if math.hypot(dx, dy) <= 20: # Check if click is within
node radius
                return node
        return None
    def highlight_node(self, node, color="yellow"):
        if node in self.nodes:
            self.canvas.itemconfig(self.nodes[node]["oval"],
fill=color)

    def clear_selection_for_edge(self):
        # Clear highlights from nodes previously selected for edge
creation
        for node in self.selected_nodes_for_edge:
            if node in self.nodes: # Ensure node still exists
                self.canvas.itemconfig(self.nodes[node]["oval"],
fill="lightblue")
        self.selected_nodes_for_edge = []

    def add_edge(self, node1, node2, weight):
        # Check if edge already exists
        for edge in self.edges:
            if set(edge["nodes"]) == set([node1, node2]):
                messagebox.showinfo("Duplicate Edge", "An edge
between these nodes already exists.")
                return

        x1, y1 = self.nodes[node1]["x"], self.nodes[node1]["y"]
        x2, y2 = self.nodes[node2]["x"], self.nodes[node2]["y"]

        line = self.canvas.create_line(x1, y1, x2, y2, width=2,
fill="black")

        # Calculate midpoint for weight display
        mid_x, mid_y = (x1 + x2) // 2, (y1 + y2) // 2

        # Calculate perpendicular offset for rectangle
        dx = x2 - x1
        dy = y2 - y1
        length = math.hypot(dx, dy)

        rect_mid_x, rect_mid_y = mid_x, mid_y # Default if length is
0
```

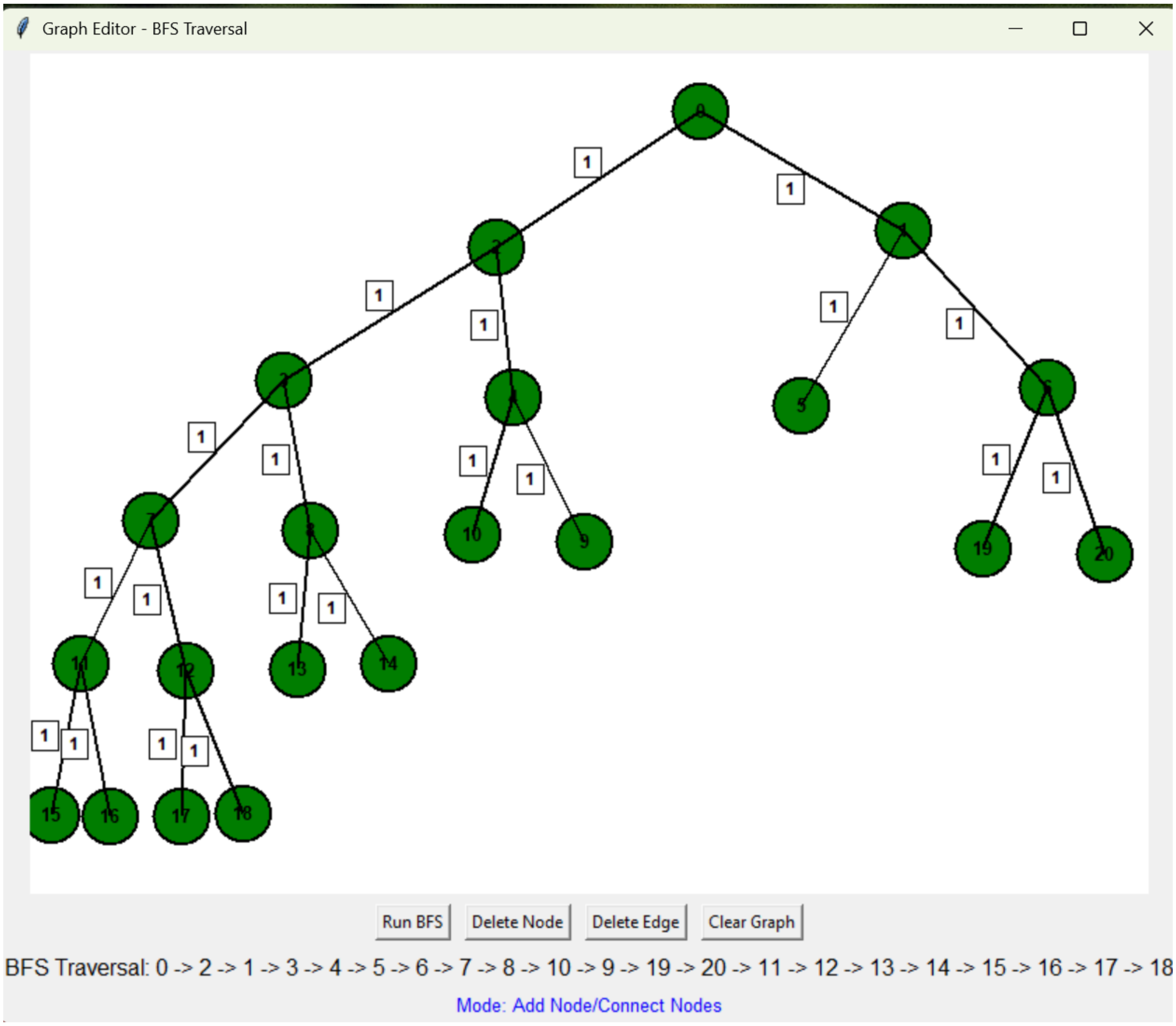
```
        if length > 0:
            nx = -dy / length
            ny = dx / length
            offset_dist = 15 # Distance to offset the rectangle from
the line
            rect_mid_x = mid_x + nx * offset_dist
            rect_mid_y = mid_y + ny * offset_dist

        text = self.canvas.create_text(rect_mid_x, rect_mid_y,
text=str(weight), fill="black", font=("Arial", 9, "bold"))
        bbox_text = self.canvas.bbox(text) # Get bounding box of the
text

        rect = None
        if bbox_text:
            pad_x, pad_y = 5, 3
            rect = self.canvas.create_rectangle(bbox_text[0]-pad_x,
bbox_text[1]-pad_y, bbox_text[2]+pad_x, bbox_text[3]+pad_y,
fill="white", outline="black", tags="weight_box")
            self.canvas.tag_lower(rect, text) # Place rectangle
behind text

        self.edges.append({"nodes": (node1, node2), "weight": weight,
"line": line, "label": text, "rect": rect})

        # Update adjacency list for both directions (undirected
graph)
        self.adjacency.setdefault(node1, []).append(node2)
        self.adjacency.setdefault(node2, []).append(node1)


    def reset_graph_elements(self):
        # Reset colors of nodes and edges
        for n_id, data in self.nodes.items():
            self.canvas.itemconfig(data["oval"], fill="lightblue",
outline="black")
            self.canvas.itemconfig(data["text"], fill="black")
        for edge in self.edges:
            self.canvas.itemconfig(edge["line"], fill="black",
width=2)
            if edge["rect"]:
                self.canvas.itemconfig(edge["rect"], fill="white",
outline="black")
            self.canvas.itemconfig(edge["label"], fill="black")
        self.bfs_label.config(text="BFS Traversal: ") # Only BFS
label for this script

    def run_bfs(self):
        if not self.nodes:
            messagebox.showinfo("BFS", "No nodes in the graph.")
            return

        self.reset_graph_elements() # Reset colors before starting

        visited = set()
        self.bfs_order = []

        start_node = list(self.nodes.keys())[0] # Start BFS from the
first available node

        queue = [start_node]
        visited.add(start_node)

        self.highlight_node(start_node, "lightgreen")
        self.root.update()
        self.root.after(500)

        while queue:
            node = queue.pop(0)
            self.bfs_order.append(node)
            self.bfs_label.config(text="BFS Traversal: " + " ->
".join(self.bfs_order))

            self.highlight_node(node, "green") # Node being processed
            self.root.update()
            self.root.after(500)

            # Find neighbors from adjacency list
            for neighbor in self.adjacency.get(node, []):
                if neighbor not in visited:
                    # Find the corresponding edge
                    for edge in self.edges:
                        # Check both directions for undirected edge
                        if (edge["nodes"] == (node, neighbor)) or
(edge["nodes"] == (neighbor, node)):
                            # Highlight current edge
                            self.canvas.itemconfig(edge["line"],
fill="red", width=4)
                            if edge["rect"]:
                                self.canvas.itemconfig(edge["rect"],
fill="yellow", outline="red")
                            self.canvas.itemconfig(edge["label"],
fill="red")
                            self.root.update()
                            self.root.after(500)

                            visited.add(neighbor)
                            queue.append(neighbor)
                            self.highlight_node(neighbor,
"lightgreen") # Mark as visited (queued)
                            self.root.update()
                            self.root.after(500)

                            # Restore edge color after traversal step
                            self.canvas.itemconfig(edge["line"],
fill="black", width=2)
                            if edge["rect"]:
                                self.canvas.itemconfig(edge["rect"],
fill="white", outline="black")
                            self.canvas.itemconfig(edge["label"],
fill="black")
                            break # Found the edge, move to next
neighbor

        messagebox.showinfo("BFS Complete", "Breadth-First Search
traversal finished.")
        # Ensure final nodes are set to final visited color (if not
already)
        for node in self.bfs_order:
             self.highlight_node(node, "green")
        self.root.update()

    def delete_node_mode(self):
        self.clear_selection_for_edge() # Clear any pending edge
selection
        self.current_mode_label.config(text="Mode: Delete Node (Click
a node)")
        self.canvas.bind("<Button-1>", self._delete_node_click)

    def _delete_node_click(self, event):
        clicked = self.get_node_at(event.x, event.y)
        if clicked:
            if messagebox.askyesno("Delete Node", f"Are you sure you
want to delete node {clicked}?"):
                # Remove associated edges from canvas and self.edges
list
                edges_to_remove = [edge for edge in self.edges if
clicked in edge["nodes"]]
                for edge in edges_to_remove:
                    self.canvas.delete(edge["line"])
                    self.canvas.delete(edge["label"])
                    if edge["rect"]:
                        self.canvas.delete(edge["rect"])
                    self.edges.remove(edge)

                # Remove node from canvas and self.nodes
                self.canvas.delete(self.nodes[clicked]["oval"])
                self.canvas.delete(self.nodes[clicked]["text"])
                del self.nodes[clicked]

                # Update adjacency list
                if clicked in self.adjacency:
                    del self.adjacency[clicked]
                for node_id in self.adjacency:
                    self.adjacency[node_id] = [n for n in
self.adjacency[node_id] if n != clicked]
```

```
            self.reset_canvas_bindings() # Reset to default mode
            self.current_mode_label.config(text="Mode: Add
Node/Connect Nodes")
        else:
            messagebox.showinfo("Delete Node", "No node clicked.
Click a node to delete it.")

    def delete_edge_mode(self):
        self.clear_selection_for_edge() # Clear any pending edge
selection
        self.current_mode_label.config(text="Mode: Delete Edge (Click
2 connected nodes)")
        self.canvas.bind("<Button-1>", self._delete_edge_click)

    def _delete_edge_click(self, event):
        clicked = self.get_node_at(event.x, event.y)
        if clicked:
            if clicked not in self.selected_nodes_for_edge:
                self.selected_nodes_for_edge.append(clicked)
                self.highlight_node(clicked, "orange")

            if len(self.selected_nodes_for_edge) == 2:
                node1, node2 = self.selected_nodes_for_edge
                edge_found = False
                for edge in self.edges[:]: # Iterate over a copy to
allow modification
                    if set(edge["nodes"]) == set([node1, node2]):
                        if messagebox.askyesno("Delete Edge",
f"Delete edge between {node1} and {node2}?"):
                            self.canvas.delete(edge["line"])
                            self.canvas.delete(edge["label"])
                            if edge["rect"]:
                                self.canvas.delete(edge["rect"])
                            self.edges.remove(edge)

                            # Update adjacency list
                            if node2 in self.adjacency.get(node1,
[]):
```

```
                                self.adjacency[node1].remove(node2)
                            if node1 in self.adjacency.get(node2,
[]):
                                self.adjacency[node2].remove(node1)
                            edge_found = True
                            break
                if not edge_found:
                    messagebox.showinfo("Delete Edge", "No edge found
between the selected nodes.")

                self.clear_selection_for_edge()
                self.reset_canvas_bindings()
                self.current_mode_label.config(text="Mode: Add
Node/Connect Nodes")
        else:
            messagebox.showinfo("Delete Edge", "Click on two nodes to
define the edge to delete.")
            self.clear_selection_for_edge() # If clicked on empty
space, clear selection

    def clear_graph(self):
        if messagebox.askyesno("Clear Graph", "Are you sure you want
to clear the entire graph?"):
            self.canvas.delete("all")
            self.nodes = {}
            self.edges = []
            self.adjacency = {}
            self.node_id = 0
            self.selected_nodes_for_edge = []
            self.bfs_order = []
            self.reset_graph_elements() # Also clears labels
            self.reset_canvas_bindings()
            self.current_mode_label.config(text="Mode: Add
Node/Connect Nodes")


    def reset_canvas_bindings(self):
        # Reset canvas bindings to default add node/connect nodes
mode
```

```
        self.canvas.bind("<Button-1>", self.mouse_click)
        self.current_mode_label.config(text="Mode: Add Node/Connect Nodes")


if __name__ == "__main__":
    root = tk.Tk()
    app = GraphGUI(root)
    root.mainloop()
```

OUTPUT:



BFS Traversal: 0 -> 2 -> 1 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 10 -> 9 -> 19 -> 20 -> 11 -> 12 -> 13 -> 14 -> 15 -> 16 -> 17 -> 18

Mode: Add Node/Connect Nodes