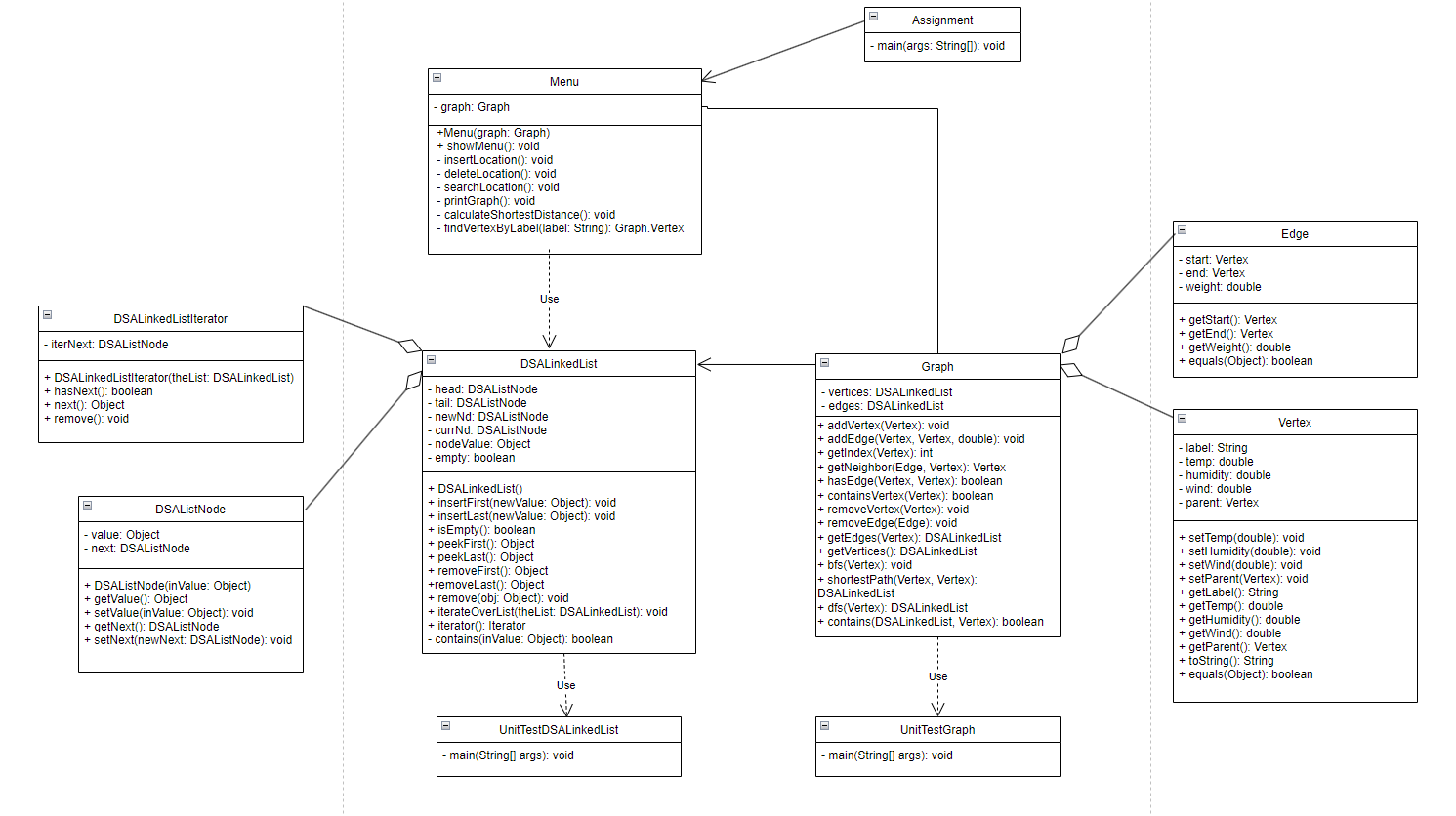
Design of the program:

The program is designed to monitor bushfires using unmanned aerial vehicles (UAVs). The UAVs collect data on temperature, humidity, wind speed, and other relevant factors to identify areas at high risk of bushfires and areas that need attention. The program consists of several classes and functionalities. I used some of my older code (DSALinkedList and Graph) and has been cited appropriately.

Data Structures/Algorithms used and Approach:

**Data structures:**

**Linked List:** Initially I was using arrays to store the vertices/edges of the Graph but soon realized it was very inefficient and created a DSALinkedList class which I would use to store the data. This made accessing the data much more simple and allowed me to create the linked list of dynamic size which would cater to the functionality of both inserting and deleting vertices. I added the remove(Object obj) method which removes a specific object from the linked list. It first checks if the list is empty and throws an exception if so. Then it checks if the object to remove is the head. If it is, it calls the removeFirst() method to remove the head. Otherwise, it iterates through the list starting from the second node (after the head) until it finds the node with the matching object. It removes the node by updating the previous node's reference to skip the current node. If the removed node is the tail, it updates the tail to the previous node. If the object is not found in the list, it throws an exception. This method is called by the Graph class when the user chooses to remove a vertex from the graph.

**Algorithms:**

**Breadth-First Search** (Shortest Path). BFS is employed to calculate the shortest distance between vertices in the graph. It starts from a specified start vertex, explores its neighbours, and continues layer by layer until all vertices are visited. This algorithm is implemented in the BFS method of the Graph class, using a queue to manage the traversal. After performing BFS, the Shortest Path Backtracking algorithm is employed to determine the actual shortest path between two vertices. The shortestPath method backtracks from the end vertex to the start vertex using the parent property set during BFS, forming a linked list representing the shortest path. These algorithms work in tandem to efficiently compute the shortest distances and paths, which find applications in route planning, network analysis, and optimization problems.

**Depth-First Search:** The Depth-First Search (DFS) algorithm in the provided code is used to traverse the graph starting from a specified vertex. The purpose of the DFS algorithm is to visit all the vertices in the graph, exploring as far as possible along each branch before backtracking. The DFS algorithm implemented in the dfs method of the Graph class follows a similar approach. It uses a stack (implemented as a DSALinkedList) to keep track of the vertices to visit. Starting from the specified start vertex, it explores one vertex at a time, moving to its neighbouring vertices until there are no unvisited neighbours left. When a dead end is reached, it backtracks by removing the last vertex from the stack and continues exploring other unvisited paths. Although I implemented the dfs method, it is not used by my menu anymore as changes I made along the way to my data structures would not accommodate its functionality. I have left it in my code with a provided test harness to show that it does work.

Class Descriptions:

**Assignment class:**

The Assignment class serves as the entry point for the program. It begins by reading input from a file called "location.txt" using a BufferedReader and FileReader to retrieve the number of vertices and edges. It then creates an instance of the Graph class and initializes an empty DSALinkedList to hold the vertices. Next, it iterates over the number of vertices and creates a new Graph.Vertex object for each vertex, assigning a label and inserting it into the vertices list. After creating all the vertices, it adds them to the graph using the addVertex method. Then, it reads edge information from the file and creates edges between the corresponding vertices using the addEdge method. Finally, it reads UAV data from "UAVdata.txt" and assigns temperature, humidity, and wind speed values to each vertex. It creates an instance of the Menu class, passing the graph to it, and invokes the showMenu method to start the menu-based interaction with the user.

**Menu class:**

The Menu class handles the user interface and provides a menu-driven system for interacting with the graph. It presents a menu with options such as inserting a location, deleting a location, searching for a location, printing the graph, calculating the shortest distance between vertices using breadth-first search (BFS), and exiting the program. Based on the user's choice, the corresponding methods are invoked to perform the desired operations.

Menu constructor: The constructor initializes a Menu object by accepting a Graph instance as a parameter and assigning it to the graph field.

showMenu method: The showMenu method displays a menu to the user using System.out.println statements and takes user input using a Scanner. It uses a while loop to repeatedly display the menu until the user chooses to exit. Based on the user's choice, it calls different methods to perform the corresponding operations.

insertLocation method: The insertLocation method prompts the user to enter the label, temperature, humidity, and wind speed of a location. It reads the user's input using a Scanner and creates a new Graph.Vertex object with the provided information. It sets the temperature, humidity, and wind speed values using the setTemp, setHumidity, and setWind methods, respectively. It then adds the vertex to the graph using the addVertex method and prints a success message.

deleteLocation method: The deleteLocation method prompts the user to enter the label of a location to delete. It reads the user's input using a Scanner and creates a new Graph.Vertex object with the provided label. It then removes the vertex from the graph using the removeVertex method and prints a success message.

searchLocation method: The searchLocation method prompts the user to enter the label of a location to search for. It reads the user's input using a Scanner and searches for the corresponding vertex in the graph using the findVertexByLabel method. If the vertex is found, it prints its label, temperature, humidity, and wind speed values. Otherwise, it prints a "Location not found" message.

printGraph method: The printGraph method retrieves the list of vertices from the graph using the getVertices method. It then iterates over each vertex and its associated edges using enhanced for loops. For each vertex, it retrieves the list of edges using the getEdges method and prints the vertex label. It then iterates over the edges and checks if the current vertex is the start or end vertex. Based on that, it prints the corresponding adjacent vertex label and edge weight.

calculateShortestDistance method: The calculateShortestDistance method prompts the user to enter the labels of the start and end locations. It reads the user's input using a Scanner and retrieves the corresponding Graph.Vertex objects using the findVertexByLabel method. It then calls the shortestPath method on the graph, passing the start and end vertices. If a shortest path is found, it prints the labels, temperature, humidity, and wind speed values of each vertex in the path, as well as the total distance. If no path is found, it prints a "No path found" message.

findVertexByLabel method: The findVertexByLabel method searches for a vertex with a specific label in the graph. It retrieves the list of vertices using the getVertices method and iterates over them to find a match. If a vertex with the specified label is found, it returns the vertex. Otherwise, it returns null.

**Graph class:**

The Graph class represents the graph data structure and provides methods for managing vertices and edges. It utilizes a custom linked list implementation called DSALinkedList to store the vertices and edges. The Graph class supports operations such as adding vertices and edges, removing vertices and edges, retrieving adjacent edges for a given vertex, checking if a vertex or edge exists in the graph, and performing breadth-first search (BFS) to find the shortest path between vertices.

The Graph.Vertex class represents a vertex in the graph and holds the label, temperature, humidity, wind speed, and a parent vertex (used for path traversal). The Graph.Edge class represents an edge in the graph, connecting two vertices and holding the weight.

Graph constructor: The constructor initializes an empty graph by creating empty linked lists for vertices and edges.

addVertex method: The addVertex method adds a vertex to the graph if it doesn't already exist. It checks if the vertex is already present in the vertices list using the containsVertex method and inserts it at the end of the list if it's not found.

addEdge method: The addEdge method adds an edge between two vertices in the graph. It creates an Edge object with the start and end vertices and the weight. It checks if the edge already exists in the edges list using the hasEdge method and inserts it at the end of the list if it's not found.

getIndex method: The getIndex method returns the index of a given vertex in the vertices list. It iterates over the vertices and compares each vertex to the given vertex until a match is found. It returns the index of the match or -1 if no match is found.

getNeighbor method: The getNeighbor method returns the neighbor vertex of a given edge and vertex. It checks if the start vertex of the edge is equal to the given vertex. If true, it returns the end vertex of the edge; otherwise, it returns the start vertex.

hasEdge method: The hasEdge method checks if an edge exists between two given vertices in the graph. It iterates over the edges and compares the start and end vertices of each edge to the given vertices. If a matching edge is found, it returns true; otherwise, it returns false.

containsVertex method: The containsVertex method checks if a vertex exists in the graph. It uses the contains method of the vertices linked list to check for the presence of the given vertex.

removeVertex method: The removeVertex method removes a vertex from the graph. It retrieves the edges connected to the vertex using the getEdges method and removes them using the removeEdge method. Then, it removes the vertex from the vertices list.

removeEdge method: The removeEdge method removes an edge from the graph. It removes the edge from the edges list.

getEdges method: The getEdges method returns the edges connected to a given vertex. It creates a new linked list and iterates over the edges. For each edge, it checks if the start or end vertex is equal to the given vertex and adds the edge to the linked list if a match is found. It returns the linked list of edges.

getVertices method: The getVertices method returns the linked list of vertices in the graph.

bfs method: The bfs method performs a breadth-first search traversal starting from a given start vertex. It uses a queue (DSAQueue) and a linked list (DSALinkedList) to keep track of visited vertices. It starts by enqueueing the start vertex and marking it as visited. It then enters a loop where it dequeues a vertex, retrieves its neighbors using the getEdges method, and enqueues any unvisited neighbors while marking them as visited by setting their parent vertex. The method visits all reachable vertices from the start vertex.

shortestPath method: The shortestPath method finds the shortest path between two vertices using breadth-first search (bfs). It calls bfs with the start vertex to populate the parent pointers of each vertex. It then backtracks from the end vertex to the start vertex using the parent pointers, creating a linked list (DSALinkedList) of vertices representing the shortest path. It returns the linked list.

The DSALinkedList class is a custom implementation of a singly linked list data structure. It provides methods for inserting and removing elements at the beginning and end of the list, as well as retrieving the first and last elements. It also includes methods for checking if the list is empty and for iterating over the elements in the list.

**DSAlinkedList class:**

The DSALinkedList class represents a linked list data structure in Java. It provides various methods for manipulating and accessing elements in the linked list. This class is used as part of the implementation for the Graph class and other operations in the given code. The DSALinkedList class uses a private inner class called DSAListNode to represent each node in the linked list. Each node contains a value and a reference to the next node in the list.

insertFirst(Object newValue): This method inserts a new element at the beginning of the list. It creates a new DSAListNode object with the given value and updates the next reference to point to the previous first node. If the list is empty, both the head and tail references are updated to point to the new node.

insertLast(Object newValue): This method inserts a new element at the end of the list. It creates a new DSAListNode object with the given value and updates the next reference of the current last node to point to the new node. The tail reference is then updated to point to the new node.

isEmpty(): This method checks if the list is empty by verifying if the head reference is null. If head is null, it means there are no elements in the list, and the method returns true; otherwise, it returns false.

peekFirst(): This method returns the value of the first element in the list without removing it. It first checks if the list is empty and throws an exception if it is. Otherwise, it retrieves the value of the head node and returns it.

peekLast(): This method returns the value of the last element in the list without removing it. It checks if the list is empty and throws an exception if it is. If not, it iterates through the list until it reaches the last node, retrieves its value, and returns it.

removeFirst(): This method removes and returns the first element from the list. It checks if the list is empty and throws an exception if it is. If not, it retrieves the value of the head node, updates the head reference to point to the next node, and returns the retrieved value. If the list becomes empty after removal, the tail reference is also updated to null.

remove(Object obj): This method removes the specified object from the list. It checks if the list is empty and throws an exception if it is. It then iterates through the list until it finds the node containing the specified object. Once found, it updates the references of the previous node to bypass the node to be removed. If the node to be removed is the head, the head reference is updated accordingly. If the node to be removed is the tail, the tail reference is updated to the previous node. If the object is not found in the list, it throws a NoSuchElementException.

iterateOverList(DSALinkedList theList): This method demonstrates how to iterate over the elements of the list using an iterator. It creates an iterator by calling the iterator() method on the list, and then iterates over the elements using a while loop and the hasNext() and next() methods of the iterator. It also keeps track of the iteration count.

iterator(): This method returns a new instance of the DSALinkedListIterator class, which implements the Iterator interface. It takes the current DSALinkedList instance as a parameter and initializes the iterNext reference to the head node.

contains(Object inValue): This method checks if the list contains a given object. It iterates over the list and compares the value of each node with the given object. If a match is found, it returns true; otherwise, it returns false.

The DSALinkedList class also has a private inner class called DSAListNode, which represents a single node in the linked list. Each DSAListNode object contains a value and a reference to the next node in the list.

Testing methodologies:

I implemented test harnesses for the DSALinkedList and Graph classes as well as a test harness for the dfs method. The test harnesses were essential to ensuring that all methods of the classes were operating as expected to reduce time spent debugging logical errors caused on my behalf. I also created a test harness for the dfs method to prove that it works and so I could test it whenever I made significant changes to my Graph class. The Assignment and Menu class do not require specific test harnesses in my opinion as the menu is a test harness for the Assignment class and the Menu itself is a form of user acceptance testing.

In terms of error handling, my user input is validated by a default switch case which made my try catch statement redundant. I have also setup the main method to catch any FileIO exceptions. Most of my error handling is implemented inside each method in the form of a default/base statement that is printed out if there are any encountered errors.

Reflection:

As mentioned earlier, implemented the DFS method within the Graph class and tested its functionality but did not make it part of the menu as I could not see its application. I already had a search and print algorithm in place so the DFS methos was not needed. I feel I may have misunderstood this part of the assignment however I discussed with my peers and they were also in a similar situation.

I could also improve on my error handling by using more try-catch blocks to catch exceptions and provide error messages to the user. My input validation could also be better, one example I can think of in hindsight is my code for inserting a location does not check if the label is unique and so it could lead to duplicate vertices (same label). I also think I could have improved on my commenting for my DSALinkedList and Graph class specifically as I find it quite hard to follow the logic the next day after writing the code. Often times I found myself breaking the code because I decide to delete variables/methods I thought aren’t being used but they are being used I just forgot where due to poor commenting/variable names. I also created a Hash table class but ended up losing a few hours of work which took me a while to restore, and I never re-implemented the hash table.

Further investigation:

Avenues for further investigation could include how other environmental factors could influence bushfires such as altitude, proximity to known hotspots and type of plant life. The investigation could also look into mapping these variables of each location multiple times over a period of time to try and see if there is a pattern.