UNIVERSITY OF ARKANSAS

CSCE 4133/5133 Algorithms

Homework Assignment #1
Linked List and Queue

Submission Deadline: Mid-Night (11:59PM) February 2, 2026

Instructions

- **Submissions:** Students can use either Google Docs or LaTeX for writing the report. The report template is provided in the Blackboard (Material/Report-template.zip). Your submission should be a zip file containing your report and the source code, including your implementation. Name the zip file as `lastname_studentID.zip`. Submissions should be made via BlackBoard.
- **Policy:** Students must start working on this assignment independently and review the late-day policy.
- If you have any questions, please contact us via Teams or email: kimtran@uark.edu and maalexra@uark.edu.

Linked List and Queue

1. Overview

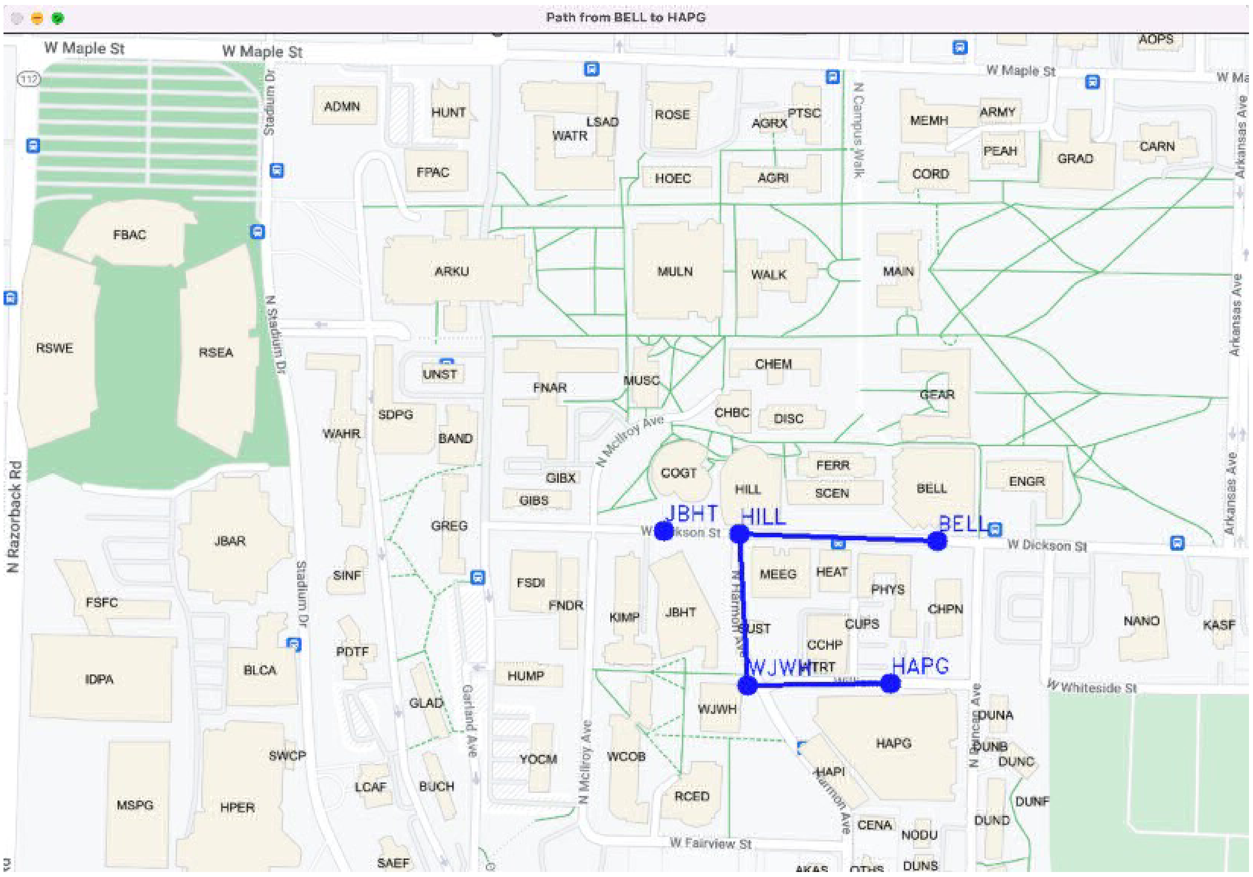

Figure 1: *Figure 1: Visualization of Map Engine*

The assignments in Algorithms are designed to implement a simple version of a map engine (*Figure 1*) that will help students to search for a path on the campus from one building to other buildings. These assignments help students to understand and use data structures and algorithms learned in the class to implement a simple practical application. This assignment is designed in **Python3**. The students are only required to write the missing implementation following the instructions and requirements. ***Graduate students may have additional requirements compared to undergraduate students.***

1.1 Homework 1: Warm-Up with Map Engine

In this homework, we will warm up by implementing a simple version of a map engine where we will find a path from one building to another building. The map is represented as a graph structure in which:

- Each node represents one building (or a landmark).
- An undirected edge between two nodes represents the path between buildings/landmarks.

A list of edges in a graph is represented by an adjacency list that is implemented by a ***linked list***. To search for a path on the graph, we use a simple version of a graph searching algorithm, specifically the Breadth

First Search (BFS) algorithm, which is implemented using a ***queue***.

In this assignment, the implementation of BFS is provided, as this algorithm will be studied in later lectures.

1.2 Requirements

You are **REQUIRED** to implement the ***Linked List*** and ***Queue*** structures. If you think the homework is too long, don't worry; it is detailed because we provide comprehensive assignment descriptions.

1.3 Hints for Success

- Start the homework early.
- Read the homework descriptions carefully before starting your implementation.
- Before implementing the required functions, try compiling the source code first. The source code with an empty implementation can be compiled successfully. This step will help you understand the procedure of each problem.
- If you forgot the linked list and queue data structures, you are encouraged to revise these structures in the lecture slides.

1.4 Source Code Organization

The source code is organized as follows:

File or Folder	**Required Implementation**	**Description**
`assets/`	No	Contains the data of homework
`linked_list.py`	Yes	Implements the linked list structure
`queue.py`	Yes	Implements the queue structure
`graph.py`	No	Implements the graph structure and algorithms
`main.py`	No	Implements the main program

https://en.wikipedia.org/wiki/Adjacency_list
We assume that each edge in a graph is unweighted in this assignment.

2. Implementation

2.1. Linked List Node and Linked List

In this section, you are going to implement a template class for the linked list structure. In this assignment, we assume the linked list contains unique values, meaning there are no two nodes with the same value.

The template class in Python3 is a class that allows us to operate with generic data types. In other words, we can use the template class of a linked list for any data type (e.g., integer, float, etc.). In this homework, there are two different requirements for undergraduate and graduate students:

- **Undergraduate Students:** You are required to implement a ***singly linked list***.
- **Graduate Students:** You are required to implement a ***doubly circular linked list***.

The linked list is defined as follows:

```python
# Template class for Linked List Node
class LinkedListNode:
    def __init__(self, value=None, next=None, prev=None):
        self.value = value
        self.next = next
        self.prev = prev
# Template class for Linked List
class LinkedList:
    def __init__(self):
        self.root = None

    def insert(self, value) -> LinkedListNode:  # Insert a new node
        # YOUR CODE HERE

    def find(self, value) -> LinkedListNode: # Find a node with a specific value
        # YOUR CODE HERE

    def remove(self, value) -> LinkedListNode: # Remove a node with a specific value
        # YOUR CODE HERE

    def size(self) -> int: # Return the number of nodes in the list
        # YOUR CODE HERE
```

The `class` of `LinkedListNode` defines the structure of a node in the linked list. This `class` contains three attributes, i.e., a value of the node, a pointer to the next node in a linked list, and a pointer to the previous node in a linked list ***(only for graduate students)***. The `class` of `LinkedList` needs to be implemented in `linked_list.py`.

The class of `LinkedList` defines the structure of the linked list. This class contains a root pointer of the linked list. There are five methods in this class: an init method, an insert method, a find method, a remove method, and a size method. The init method of this class is already provided, where it initializes the empty linked list. You are required to implement the four other methods in `linked_list.py`:

- `Insert`: Insert a new node containing the given value and return a pointer to the new node. If the given value already existed, return the pointer containing the given value.
- `Find`: Find a node containing the given value and return a pointer to the found node. If the given value does not exist, return the `NULL` pointer.
- `Remove`: Remove a node containing the given value and return a root pointer of a linked list. If there is no node containing the given value, simply return a root pointer.
- `Size`: Return the number of nodes in the linked list.

2.2 Queue

In this section, we are going to implement the class of a queue. The queue class is defined as follows:

```python
class QueueNode:
    def __init__(self, value=None, next=None, prev=None):
        self.value = value
        self.next = next
        self.prev = prev

class Queue:
    def __init__(self):
        self.head = None
        self.tail = None

    def empty(self) -> bool:
        # YOUR CODE HERE

    def pop(self) -> int:
        if self.empty():
            raise IndexError("Queue is empty")
        # YOUR CODE HERE
        return value

    def push(self, value):
        node = QueueNode(value, None, None)
        # YOUR CODE HERE
```

The `class` of `QueueNode` defines a node of the queue. This `class` contains three attributes: a value of the node, a pointer to the next node in the queue, and a pointer to the previous node in the queue. The implementation of this class can be found in `queue.py`.

The class of `Queue` defines the structure of the queue. This class contains two private attributes: a head pointer and a tail pointer of the queue. In expectation, we are going to push a new node to the tail of the queue and pop out a value from the head of the queue. There are four methods in this class: an init method, a push method, a pop method, and an empty method. The init method is already provided, which initializes the empty queue. You are required to implement the three other methods in `queue.py`:

- `Push`: Push a new node into the tail of a queue.
- `Pop`: Return a value in the head of the queue and remove this head node out of the queue. We assume the `pop` function is only called when the queue is not empty.
- `Empty`: Return true if the current queue is empty; otherwise, it is false. The queue is empty if and only if both the head and tail of the queue point to the `NULL` pointer.

2.3 Graph

In this section, we will detail the definition of the graph class and its purpose. The class of `Graph` is defined as follows:

```
class Graph:
    def __init__(self, n):
        self.n = n #  Number of vertices
        self.e = [LinkedList() for _ in range(n)] # Adjacency list

    def insert_edge(self, u, v, directed=False):
    def search(self, start, destination):
};
```

The class `Graph` consists of two private attributes: the number of nodes (vertices) of the graph and the adjacency list. The adjacency list is implemented by your linked list. There are three methods:

- An init method (constructor) that creates a new graph with the given number of nodes.
- An `insert_edge` method that adds one edge to the graph.
- A `search` function that finds a path between two given nodes.

The implementation of the class `Graph` can be found in `graph.py`. In this homework, the graph implementation is provided for visualization purposes only.

2.4 Main Program

The main program is fully implemented in `main.py`. This program contains the implementation of the unit tests for each module in your implementation (i.e., linked list and queue). This program also contains the visualization of the search results on the campus map.

3. Compilation and Testing

To compile and run this homework, you are required to install the following packages:

- **Python3**: This is the python3 compiler.
- **OpenCV Library**: This is used to visualize your results. You can install it by "pip install opencv-python".

If you run the program successfully and your implementation is correct, you should pass all the unit tests designed in the main program. Otherwise, the program will yield an announcement indicating which module is incorrect.

4. Submission

Write a report detailing the key ideas for implementing each required function and include the implemented functions themselves. You must also include a screenshot of the final output (from either the command prompt or terminal) after running the program.

Write your full name, email address, and student ID in the report. Your submission should be a zip file containing your report and the source code, including your implementation. Name the zip file as `lastname_studentID.zip`. Submission should be made via BlackBoard.