CSCE4133/5133 – Algorithms
Fall 2023

Assignment 1
Linked List and Stack

Out Date: Sep. 1, 2023
Due Date: **Sep. 15, 2023**

Instructions:

- **Written Format & Template:** Students can use either Google Doc or Latex
- Write your full name, email address and student ID in the report.
- Write the number of "Late Days" the student has used in the report.
- Submission via BlackBoard
- **Policy:** Can be solved in groups (two students per group at max) & review the late-day policy.
- **Submissions:** Your submission should be a PDF for the written section & a zip file with your source code implementation.
- For more information, please visit course website for the homework policy.

Linked List and Stack

1. Overview

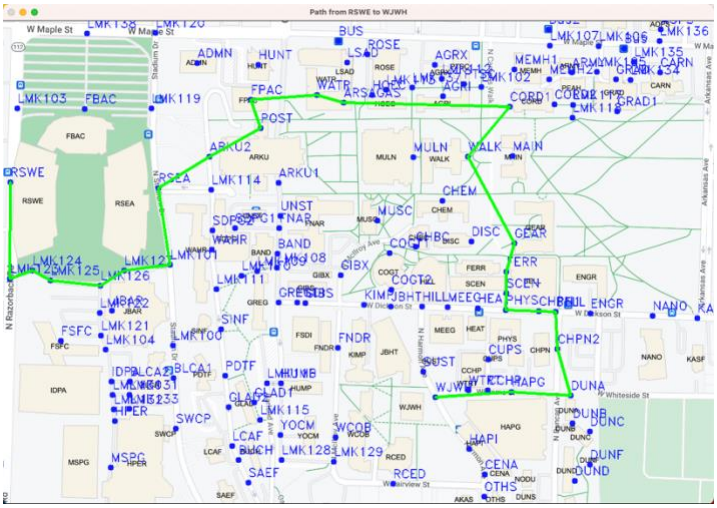

Figure 1: Visualization of Map Engine

The assignments in Algorithms are designed to implement a simple version of a map engine (Figure 1) that will help students to search 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. In the assignments of this class, students are ***REQUIRED*** to implement in C++. The students are only needed to write the missing implementation following the instructions and requirements. ***The graduate students may have more requirements than undergrad students.***

In this homework 1, we are going to warm up with a simple version of 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), and 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[1] that is implemented by a ***linked list***. To search a path on the graph[2], we use a simple version of a graph searching algorithm which is a Depth First Search (DFS) algorithm implemented by a ***stack***. In this assignment, we provide you the implementation of DFS because this algorithm will be studied in the later lectures.

In this assignment, you are ***REQUIRED*** to implement ***Linked List*** and ***Stack*** structures. If you think the homework is too long, don't worry, it is long because we provide the assignment descriptions in detail. There are some hints that may help students to succeed in this homework:

- You should start the homework early.
- You should read the descriptions of homework carefully before you start working on the homework.

[1] https://en.wikipedia.org/wiki/Adjacency_list

[2] We assume that each edge in a graph is unweighted in this assignment.

- Before implementing the required functions, you should try to compile the source code first. Basically, the source code with an empty implementation can be compiled successfully. This step will help you to understand the procedure of each problem.
- If you forgot the linked list and stack data structures, you are encouraged to revise these structures in the lecture slides.

The source code is organized as follows:

File or Folder	***Required Implementation***	***Description***
assets/	No	Contain the data of homework
include/	No	Contain headers files
src/	No	Contain source code files
bin/	No	Contain an executable file
include/linked_list.hpp	No	Define the linked list structure
include/stack.hpp	No	Define the stack structure
include/graph.hpp	No	Define the graph structure
src/linked_list.cpp	Yes	Implement the linked list structure
src/stack.cpp	Yes	Implement the stack structure
src/graph.cpp	No	Implement the graph structure and algorithms
src/main.cpp	No	Implement the main program
Makefile.windows	No	Define compilation rules used on Windows
Makefile.linux	No	Define compilation rules used on Linux/Mac OS

2. Implementation

a. 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 are assuming the linked list contains unique values. It means there are no two pointers containing the same value. The template class in C++ 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 undergrad and graduate students.

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

The header of linked list is defined (*include/linked_list.hpp*) as follows:

```
template<class T>
struct LinkedListNode {
    T value;
    LinkedListNode<T> *next;
    LinkedListNode<T> *prev;
    LinkedListNode(T value = 0, LinkedListNode<T>* next = NULL, LinkedListNode<T> *prev = NULL);
    ~LinkedListNode();
};

template<class T>
class LinkedList {
    private:
        LinkedListNode<T> *root;
    public:
        LinkedList();
        ~LinkedList();
        LinkedListNode<T>* insert(T value);
        LinkedListNode<T>* find(T value);
        LinkedListNode<T>* remove(T value);
        int size();
};
```

The struct of ***LinkedListNode*** defines the structure of a node in the linked list. This struct contains three attributes, i.e. a value of the node, a pointer to the next node in a linked list, a pointer to the previous node in a linked list (***only for graduate students***). There are two methods in this struct, i.e., a constructor (with default values) and a deconstructor. The implementation of this struct can be found in *src/linked_list.cpp*.

The class of ***LinkedList*** defines the structure of the linked list. This class contains one private attribute, i.e., a root pointer of the linked list. There are six methods in this class, i.e., a constructor, a deconstructor, an insert method, a find method, a remove method, and a size method. The constructor of this class is already provided where it initializes the empty linked list. You are required to implement the five other methods in *src/linked_list.cpp*:

- *Deconstructor:* Delete and release memory of all nodes.
- *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.

```
template<class T>
LinkedList<T>::~LinkedList() {
    // YOUR CODE HERE
}
template<class T>
LinkedListNode<T>* LinkedList<T>::insert(T value) {
    // YOUR CODE HERE
}
template<class T>
LinkedListNode<T>* LinkedList<T>::find(T value) {
    // YOUR CODE HERE
}
template<class T>
LinkedListNode<T>* LinkedList<T>::remove(T value) {
    // YOUR CODE HERE
}
template<class T>
int LinkedList<T>::size() {
    // YOUR CODE HERE
}
```

b. Stack

In this section, we are going to implement the template of a stack structure. The stack structure is defined (*include/stack.hpp*) as follows:

```
template<class T>
struct StackNode {
    T value;
    StackNode<T> *next;
    StackNode<T> *prev;

    StackNode(T value, StackNode<T> *next, StackNode <T> *prev);
```

```
    ~StackNode();
};
template<class T>
class Stack {
    private:
        StackNode<T> *head;
        StackNode<T> *tail;
    public:
        Stack();
        ~Stack();
        bool empty();
        T pop();
        void push(T value);
};
```

The struct of ***StackNode*** defines a node of the stack. This struct contains three attributes, i.e. a value of the node, a pointer to the next node in the stack, a pointer to the previous node in the stack. There are two methods in this struct, i.e., the constructor (with default value) and the deconstructor. The implementation of this structure can be found in *src/stack.cpp*.

The class of ***Stack*** defines the structure of the . This class contains two private attributes, i.e., a head pointer and a tail pointer to of the stack. In expectation, we are going to push a new node to the tail of the stack and pop out a value from the head of the stack. There five methods in this class, i.e., a constructor, a deconstructor, a push method, a pop method, and an empty method. The constructor of this class is already provided in which it initializes the empty stack. You are required to implement four methods in *src/stack.cpp*:

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

```
template<class T>
Stack<T>::~Stack() {
    // YOUR CODE HERE
}
template<class T>
bool Stack<T>::empty() {
    // YOUR CODE HERE
```

```
}
template<class T>
T Stack<T>::pop() {
    T value = this->tail->value;
    // YOUR CODE HERE
    return value;
}
template<class T>
void Stack<T>::push(T value) {
    StackNode<T> *p = new StackNode<T>(value, NULL, NULL);
    // YOUR CODE HERE
}
```

c. Graph

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

```
class Graph {
    private:
        int n; // Number of vertices
        std::vector<LinkedList<int> > e; // Adjacency list
    public:
        Graph(int n);
        ~Graph();
        void insertEdge(int u, int v, bool directed = false);
        std::vector<int> search(int start, int destination);
};
```

The class **Graph** consists of two private attributes: i.e., the number of nodes (vertices) of the graph and adjacency list. The adjacency list is implemented by your linked list. There are four methods, i.e., a constructor creates a new graph with the given number of nodes, a deconstructor, an insert edge method adds one edge into a graph, and a search function finds a path between two given nodes. The implementation of the class Graph can be found in *src/graph.cpp*.

d. Main Program

The main program is fully implemented in *src/main.cpp*. This program contains the implementation of the unit test of each module in your implementation (i.e., linked list and stack). 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 some following packages:

- **Make.** This is to execute rules to compile your program.
- **GCC/G++.** This is the C++ compiler.
- **OpenCV Library** (Optional)**.** This is used to visualize your results.

These tools are all supported on Windows, Linux, and Mac OSX. There will be a document and a tutorial to install these tools in our class.

There are two files named ***Makefile.windows*** and ***Makefile.linux***. If you are using Windows, rename ***Makefile.windows*** to ***Makefile***. If you are using Linux or Mac OSX, rename ***Makefile.linux*** to ***Makefile***. If you do not want to use OpenCV for visualization, open file ***Makefile*** and edit line ***OPENCV=1*** to ***OPENCV=0.*** Please noted that if you are using OpenCV 4+, you may have to edit lien ***OPENCV4=0*** to ***OPENCV4=1***. To compile the source code, you open the terminal and go to the source code folder and type **make compile** (or **mingw32-make compile**)**.** If you compile successfully, it will give you no errors and output a folder named bin that contains the executable file.

To run and test your implementation, using the terminal and type **make run** (or **mingw32-make run** on Windows). If you run successfully and your implementation is correct, you should pass all the unit tests designed in the main program. Otherwise, it will yield you an announcement indicating which module is not correct.