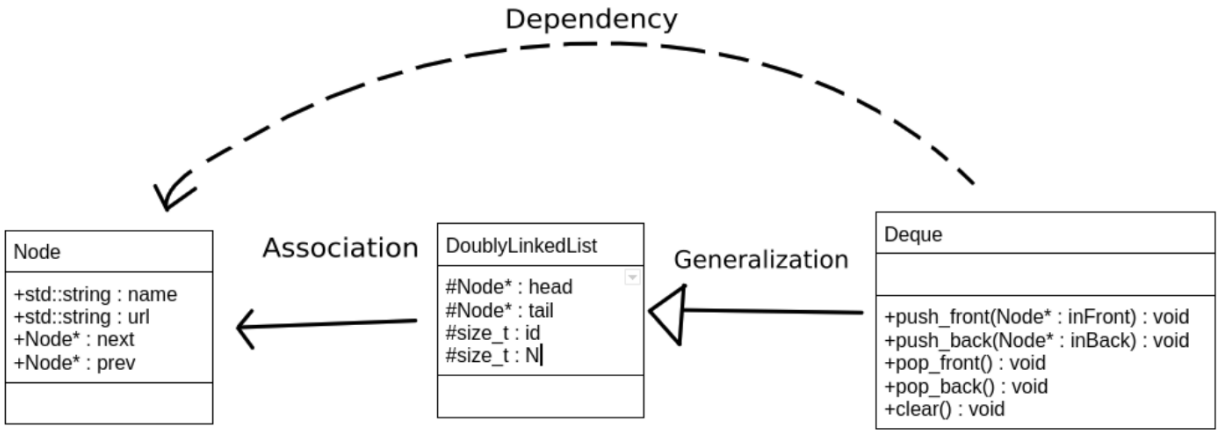
1. Overview of Classes

What classes did you design: I designed a **Node**, **DoublyLinkedList** and **Deque** class

The role of each class: The **Node** class acts as a container to store URLs and URL-names as well as act as a node a part of the **DoublyLinkedList** class which is a doubly linked list. **Deque** inherits from that class and adds member functions to increase the functionality and simulate a browser history.

How do your classes relate: The **Deque** class inherits from the **DoublyLinkedList** class and the **Node** class is a dependency to the **DoublyLinkedList** and **Deque** class.

2. UML Class Diagram



3. Details on Class Decisions

- Node class
  - Node class has data members **name**, **url** to store URLs and URL-names. Since it is a node in a doubly linked list it has pointers to its **next** and **prev** nodes.
- Design decisions regarding constructors and destructors for Node
  - The constructor simply sets the **name** and **url** to passed values and **next** and **prev** to nullptr. This is because **next** and **prev** will later be overwritten and if they are accessed before this, they will simply be nullptr instead of *possibly* being a random number (depends on compiler)
  - The deconstructor simply sets the strings to empty strings and the pointers to nullptr. This is not necessary as string is a class and will delete itself upon leaving scope.

- DoublyLinkedList Class
  - Has protected data members **head**, **tail**, **id**, **N** to keep track of the head, tail, size, and max size respectively of the doubly linked list.

- Design decisions regarding constructors and destructors for DoublyLinkedList
    - The construct simply sets its data members to its passed in values. The deconstruct just sets all pointers to nullptr.

- Deque Class
    - Inherits data members from the DoublyLinkedList class, but has public functions to handle inputted commands
    - I also made the getters for the class (**size**(), **front**(), **back**()) all const functions as they do not need to change any objects
- Design decisions regarding constructors and destructors for Deque
    - Since Deque inherits DoublyLinkedList's data members I also inherited its constructor. The deconstructor traverses the nodes from head to tail deleting each node after passing it.

4. Test Cases

I tested the possible cases; memory leaks, code functionality (e.g. popping and empty deque), and wrote a script to go through many test cases.

5. Performance Decisions

- O(n) functions
    - **clear** and **print** both traverse the deque from head to tail deleting and printing each node respectively
    - **find** traverses the deque from head to tail till it finds a node with the same name as the passed in name and prints it.
- O(1) functions
    - **m** calls the constructor which is O(1)
    - **push_back** and **push_front** both are O(1) as in the worst case they pop a node and then push a node to the front which are both O(1)
    - **pop_back** and **pop_front** are both O(1) as they are deletion in a doubly linked list (which is O(1))
    - **size** is a getter which simply returns data member **id** which keeps track of the number of elements in the deque as we add or delete items (is O(1))
    - **front** and **back** are also getters which return the **head** and **tail** respectively (is O(1))