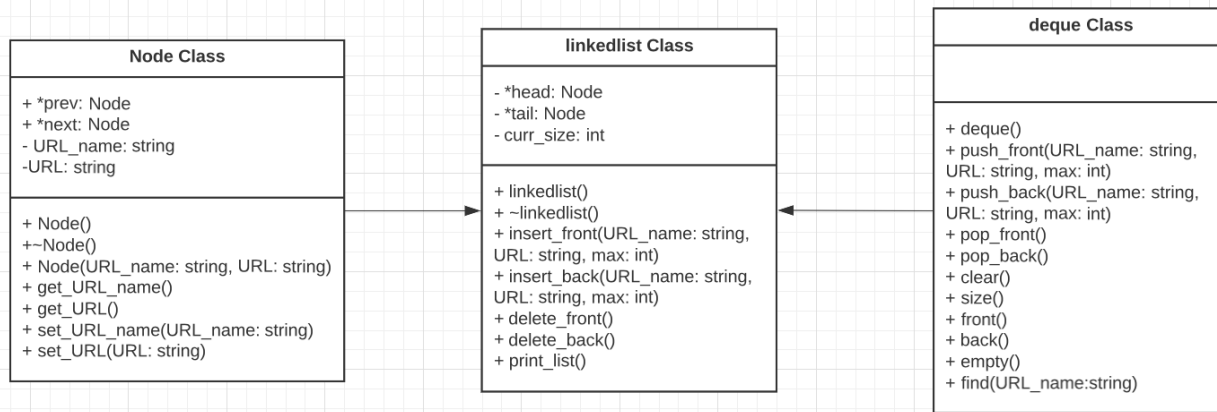


2. UML Class diagram



3. Details on design decisions

The constructor for class Node creates an object that contains empty strings and nullptrs.

The destructor for class Node sets the pointers next and prev to nullptr.

The constructor for class linkedlist sets the objects head and tail of the class Node to nullptr.

The destructor for class linkedlist deletes all the nodes in the linked list

The constructor for class Song creates an object that contains the song name and artist name

The constructor for class Playlist dynamically allocates an array of objects from the class song.

The destructor for class Playlist deletes the dynamically allocated array

4. Test cases

Test 1: make a linkedlist `m 3` and then force shut the program `exit`

Test 2: make a linkedlist `m 3`, add 3 nodes `push_front "firefox" "www.firefox.com"` `push_front "google" "www.google.ca"`; `push_front "safari" "www.safari.ca"`, print the linked list to ensure nodes are created and ordered (remember it prints back to front) `print`

Test 3: make a linkedlist `m 3`, add 3 nodes `push_front "firefox" "www.firefox.com"` `push_front "google" "www.google.ca"`; `push_front "safari" "www.safari.ca"`, delete the first node `pop_front`, delete the last node `pop_back`, print to ensure they were deleted `print`

Test 4: make a linkedlist `m 3`, add 3 nodes `push_front "firefox" "www.firefox.com"` `push_front "google" "www.google.ca"`; `push_front "safari" "www.safari.ca"`, print the first node `front`, delete the first node `pop_front`, print the first node again to ensure the head was changed after the first node was deleted `front`, force shut the program `exit`

Test 5: make a linkedlist `m 3`, check if empty `empty`, add 3 nodes `push_front "firefox" "www.firefox.com"` `push_front "google" "www.google.ca"`; `push_front "safari" "www.safari.ca"`, check the size of the linked list `size`, delete all the nodes at once `clear`, check the size again to ensure all the nodes were deleted `size`, force shut the program `exit`

Test 6: make a linkedlist `m 3`, add 3 nodes `push_front "firefox" "www.firefox.com"` `push_front "google" "www.google.ca"`; `push_front "safari" "www.safari.ca"`, check the size of the linked list `size`, delete all the nodes at once `clear`, check the size again to ensure all the nodes were deleted `size`, force shut the program `exit`

Test 7: make a linkedlist `m 3`, add 3 nodes `push_front "firefox" "www.firefox.com"` `push_front "google" "www.google.ca"`; `push_front "safari" "www.safari.ca"`, check the size of the linked list `size`, add another node `push_front "translate" "www.googletranslate.ca"`, print the linked list to ensure firefox was removed `print`, search for google `find "google"`, force shut the program `exit`

5. Performance Considerations

Insertions (`push_front`, `push_back`) - $O(1)$ - these algorithms have a constant run time because the size of the input has no effect on it. It will always check if the linked list is at its max - $O(1)$, then it has two options. It can call `insert_front` - which will create a new node and place it at the beginning of the linked list - $O(1)$. Or it can call `delete_back` - which will just delete the last node of the linked list - $O(1)$ and then it will call `insert_front` - $O(1)$. Since each step in this algorithm has a constant run time, the run time of the algorithm is $O(1)$

Deletions (`pop_front`, `pop_back`) - $O(1)$ These algorithms have a constant run time because every instruction it completes always requires a constant run time regardless of the input size. It always first checks if the linked list is empty - $O(1)$. Then it deletes the first/last node - $O(1)$. Since each step in this algorithm has a constant run time, the run time of the whole algorithm is $O(1)$

Clear - $O(n)$ This algorithm has a linear runtime. It recursively calls the `delete_front` function, which has a constant run time, but as the size of the linked list increases, the number of times `delete_front` is called increases, so if there are n nodes, it will be called n times - $O(n)$

Size - $O(1)$ This algorithm has a constant run time. It returns the value of the current size of the linked list - $O(1)$

Front - $O(1)$ This algorithm has a constant runtime. It returns the data that is at the first node - $O(1)$

Back - $O(1)$ This algorithm has a constant runtime. It returns the data that is at the last node - $O(1)$

Empty - $O(1)$ This algorithm has a constant runtime. It checks if the current size is 0 - $O(1)$

Find - $O(n)$ This algorithm has a linear runtime. It has to iterate through the linked list until it finds the node the user asked for. The worst case is that the node is not in the linked list and it has to iterate to the end of the list - $O(n)$. So as the input size (linked list size) increases, the runtime also increases because it will iterate through more nodes.

Print - $O(n)$ This algorithm has a linear runtime. It has to iterate through the linked list and print all the nodes. As the linked list size increases, the algorithm will have to do more iterations, so the runtime depends on the size of the input - $O(n)$

Exit - $O(n)$ This algorithm has a linear runtime because at the time of exit, the algorithm has to call `clear` to delete all the nodes, and since `clear` has a linear runtime, so does exit.