ECE250 – Project 1
Dynamic Deques - WebBrowser Design Document

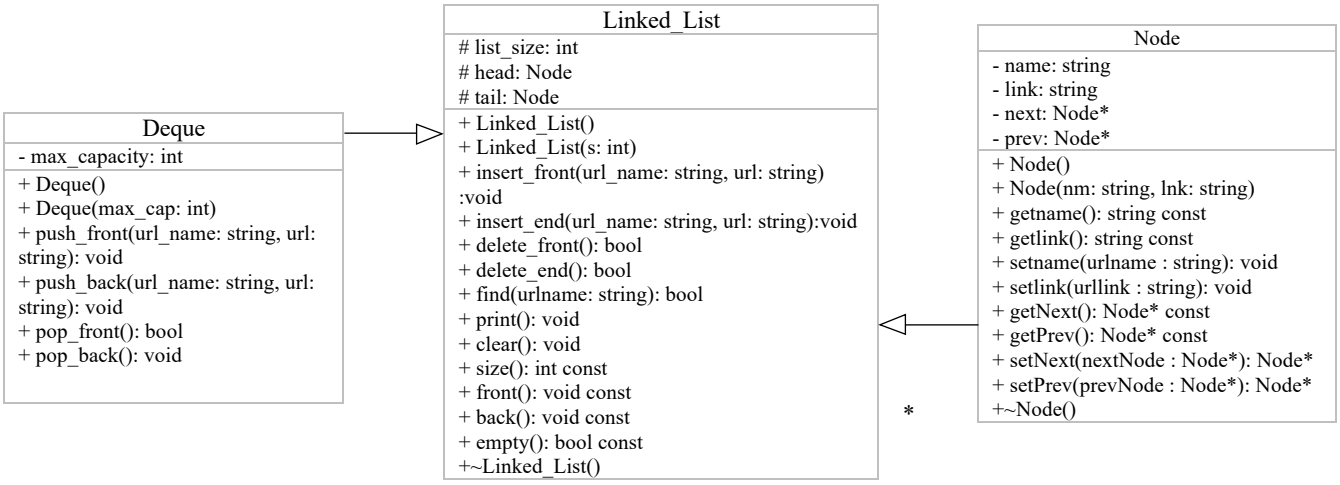
Nandita Lohani, UW UserID: nlohani

October 18, 2022

1. Overview of classes

Class	Node	Linked_List	Deque
Description	This class represents each node in the linked list. It holds the data of each node, the references to the next and previous nodes and provides getter and setter operations for the variables.	This class is a typical linked list and provides all the necessary operations such as insert at the front, insert at the end, delete, print, etc. It represents a dynamic list of nodes which are connected and referenced with pointers.	This class inherits from the Linked_List class and represents a doubly linked list. It provides additional functions on its own such as pushing nodes at the front or back and deleting at the front or back.
Member Variables	name – URL name (string) link – URL (string) *next – next node in the linked list (Node) *prev - previous node in the linked list (Node)	list_size – the size of the linked list *head – a pointer to the first node of the linked list (Node) *tail - a pointer to the last node of the linked list (Node)	max_capacity – the maximum size of the linked_list
Member Function	setname – passes a string parameter and sets it to name setlink- passes a string parameter and sets it to link setNext – passes a Node as parameter and sets it to next (Node) setlink- passes a Node parameter and sets it to prev (Node) getname – returns the name (string) getlink – returns the link (string) getNext – returns the next node (Node) getPrev – returns the previous node (Node)	size - returns the list size (int) front- outputs the data from the first node of the list back- outputs the data from the last node of the list empty - checks if the list is empty (bool) insert_front - inserts user passed parameters as data into the front of the linked list insert_end - inserts user passed parameters as data into the end of the linked list delete_front - deletes the first node (bool) delete_end - deletes the last node (bool) clear - deletes all nodes in the linked list with proper memory deallocation find - searches for the user passed URL name in the linked list (string) print - prints the values of all nodes in the list	push_front - calls insert_front from the linked list and various functions such as size and delete-end to make sure there is enough space and to make new space before inserting the data to the front push_back- calls insert_end from the linked list and various functions such as size and delete-front to make sure there is enough space and to make new space before inserting the data to the back pop_front - calls the delete_front function to remove the node at the front of the list pop_back - calls the delete_end function to remove the node at the back of the list

2. UML class diagram



3. Details on design decisions

All classes except for the Deque class in the design include two constructors to make the program more flexible by allowing the user to call it in two ways as one constructor can assign default values to the variables and the second one only accepts user-defined values. The Node, Linked_list and the Deque class all consist of a constructor that initializes the variables in the class with default values. The second constructor of the Node class assigns the values with user passed parameters and sets pointers to null. The second constructor of the Linked_List class initializes the list size to 0 and sets pointers to null.

The destructors for each class were designed to deallocate memory and to clean up for each class objects. The Node class creates pointers, the destructor sets them to nullptr to avoid any dangling pointers. Since the Linked_List class contains several pointers and nodes for the linked list, the destructor iterates through the list to manually delete each node and set the pointers to null. The Deque class inherits the linked list deconstructor.

The getter functions in the Node class and the accessors in the Linked_List (size, empty, front back) are constant functions as they only retrieve the appropriate values, they should not modify any data in the process. The print function is also constant as it should not modify anything.

4. Performance Considerations

All functions have a runtime of O(1) except for find, clear and print, which are O(n). The O(n) functions require iteration through the entire linked list and since there are reference to the next and previous nodes, the iteration is done once or linear. The constant runtime functions do not require any iteration since the head and tail pointers are readily available as well as the next and previous node references. These functions simply require changing the pointers to the correct nodes or nullptr after each operation.

1. Test cases

The testing strategy for the design was to test each section of the program before testing the program altogether. Testing and extracting the proper commands from the command line was done with various print statements within the driver file. The test for the functions and operations were tested as follows:

Test Areas	Test Cases
Creation	• Create the playlist array with user given size
insert_front, insert_end, push_front, push_back	• Add a node to front of a list • Add node to back of the list • Add more entries than the size of the list (should automatically create more space through delete) • Add a node on an empty list
delete_front, delete_end, pop_front, pop_back	• Delete a node at the front of the list • Delete a node at the back of the list • Attempt to delete a node of an empty list and consider dangling pointers • Attempt to delete front or end the only node of a list
print, clear	• Print the list from back to front • Clear the list (call print after clear to ensure everything is deleted) • Attempt to clear or print an empty list • Account for dangling pointers or memory leaks for clear function
front, back	• Return first node of the list • Return last node of the list • Attempt to return front or back of an empty list • Use front or back on the only node of the list
find	• Finds the correct node • Attempt to find a URL name that does not exist in the list • Attempt to use find on an empty list
Other	• Create a list of max size 0 and attempt to add nodes • Create a list of a large size and add many nodes to ensure program does not crash • Each function delivers the correct outputs for each case (i.e. success, found URL name URL, etc.)

In addition to the tests, there were additional tests performed to ensure all functions work altogether and there are no dangling pointers or memory leaks through Valgrind. For example, a test case file was made with all the commands and each scenario.