ECE 250 - Project 1
Dynamic Deques
Design Document

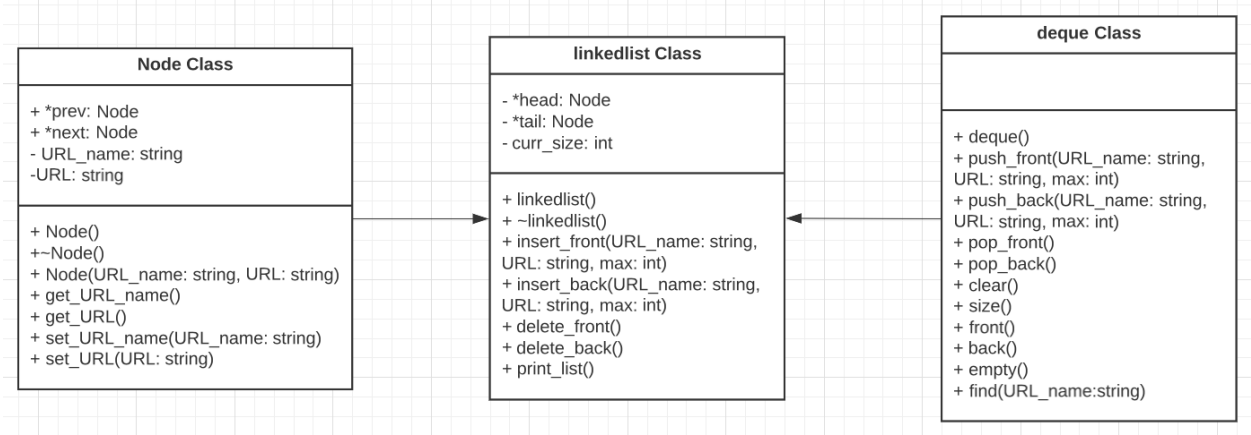
Julia Aziz, UW User ID: ja2aziz
Oct 18, 2022

Number of Pages: 2
Document name: ja2aziz_design_p1.pdf

1. Overview of classes

Class: Node
Description: Holds the information of each node. Each object of this class holds the URL name and the URL. These objects are used in the class "linkedlist" and "deque" to execute different commands requested by the user, such as adding, deleting, or finding URLs.
Member variables:
URL_name - stores the name of the URL, for example "google"
URL - stores the URL, for example "www.google.ca"
*prev - stores the pointer to the previous node, for example if the linked list is currently holding firefox->google->safari, prev of "google" would point to "firefox"
*next - stores the pointer to the next node, for example, if the linked list is currently holding firefox->google->safari, next of "google" would point to "safari"
Member functions:
Node - creates an object of the class Node and sets next and prev to nullptr and URL_name and URL to empty strings
Node - creates an object of the class Node and assigns the data given by the user to URL_name and URL
~Node - sets the pointers next and prev to nullptr
get_URL_name - returns the name of the URL since URL_name is a private member variable
get_URL - returns the URL since URL is a private member variable
set_URL_name - sets the name of the URL since URL_name is a private member variable
set_URL - sets the URL since URL is a private member variable
Class: linkedlist
Description: Fully implemented doubly linked list class that runs commands like insert, delete and print.
Member variables:
*head - an object of the class Node that is the head of the linked list
*tail - an object of the class Node that is the tail of the linked list
Curr_size - an integer that keeps track of how many URLs are added and removed
Member Functions:
linkedlist - sets the head and tail to nullptr
~linkedlist - deletes all the nodes in the linked list
insert_front - adds a node at the front of the linked list
insert_back - adds a node at the end of the linked list
delete_front - deletes the first node of the linked list
delete_back - deletes the last node of the linked list
print_list - prints all the nodes of the linked list
Class: deque
Description: This class inherits from the linkedlist class and executes user indicated commands such as pushing, popping and retrieving information
Member variables: This class has no member variables since it uses those from the linkedlist class
Member Functions:
deque - empty constructor - calls the linkedlist class constructor
push_front - calls insert_front from the linkedlist class to add a node at the front of the linked list and deletes the last node by calling delete_back if the linked list is full
push_back - calls insert_back from the linkedlist class to add a node at the back of the linked list and deletes the first node by calling delete_front if the linked list is full
pop_front - calls delete_front from the linkedlist class to delete the first node of the linked list
pop_back - calls delete_back from the linkedlist class to delete the last node of the linked list
clear - recursively calls delete on the nodes until curr_size is 0
size - returns the current size of the linked list
front - returns the first node of the linked list
back - returns the last node of the linked list
empty - checks and returns if the linked list is empty
find - iterates through the linked list to find the node that the user requested, returns found or not found

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.