ECE 250 - Project 1
Deque Class Design Document

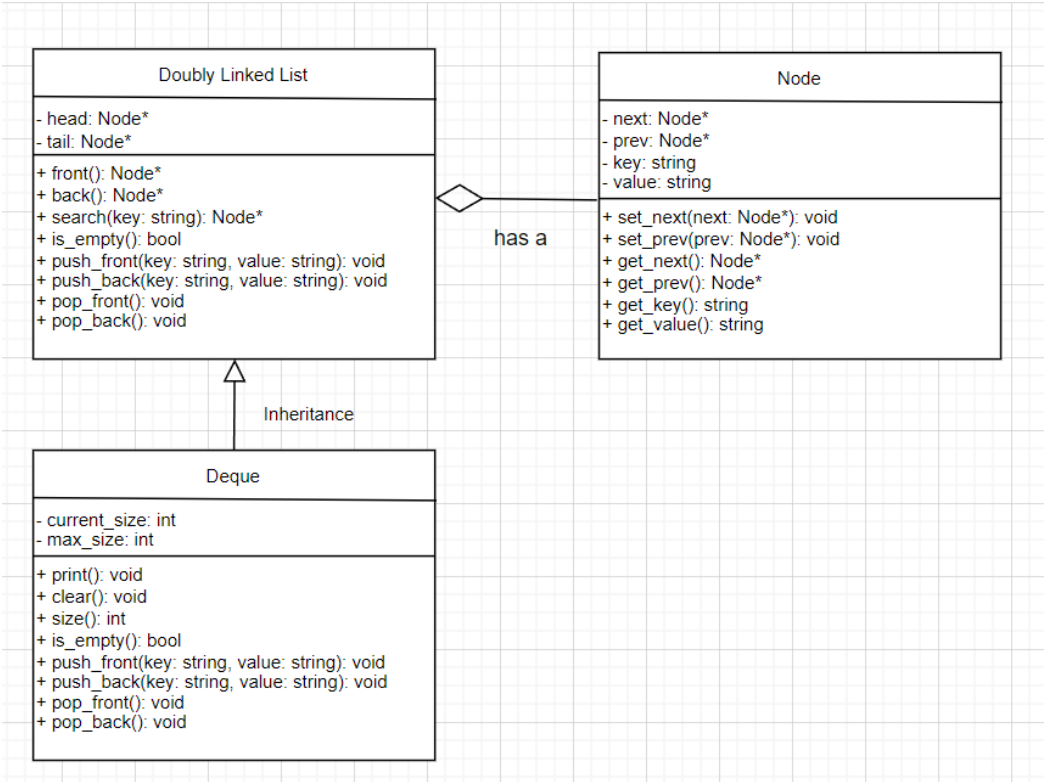
Abdullah Abdullah, UW UserID: a55abdul

Oct 18th, 2022

Overview of Classes

Class: Deque
Description: A child of the doubly linked list class that represents a deque. It uses a doubly linked list to hold data and utilises the push and pop functions to modify the data accordingly. The deque is very similar to the doubly linked list class with the main difference being that the deque has a max size attribute and specific instructions for when that max size is reached. Thus it inherits a lot of functionality from the doubly linked list class.
Class: Doubly Linked List
Description: Class that represents a doubly linked list and implements most ADT functions defined for a doubly linked list. These functions include inserting at the front, inserting at the back, deleting from the front, deleting from the back and searching for specific nodes based on their key. It is the parent of the deque class and has member variables from the Node class. The class's main purpose is to define the structure for the deque class.
Class: Node
Description: Node that holds a key and value and is used in the linked list. Each node has a next and previous Node pointer to comply with the doubly linked list. These Nodes are dynamically allocated when created for use in the doubly linked list class and must also be deallocated by the doubly linked list to avoid memory conflicts. The nodes are used to hold our data in our deque/doubly linked list.

UML Class Diagrams



Details on Design Decisions:

- **Constructor:** My node class's constructor takes in a key and value which are both strings and initialises those member variables. It also sets the next and prev pointers to nullptr to avoid pointing to random data. My doubly linked list constructor takes no parameters and assigns the head and tail pointers to nullptr for similar reasons to the node class. My deque constructor calls its parent's (doubly linked list) constructor and also sets the current size to 0. It also takes in a max size and initialises that member variable. The current size represents how many nodes are in the deque at any given time which is important to know when I have exceeded the max size and need to pop.
- **Destructor:** The destructor for the node class sets the next and prev variables to nullptr to avoid dangling pointers. The destructor for the doubly linked list class clears the linked list by repeatedly calling pop_back until the list is empty. The pop_back/push_back function deallocates memory for each Node (which is dynamically allocated on push_front/push_back) and reassigns the head and tail pointers. As a result, by repeatedly calling the pop_back function until the list is empty I know I am deallocating all the memory used. I also know that head and tail are set to nullptr so I don't need to worry about dangling pointers. My deque class inherits the destructor from the linkedlist class and doesn't need its own.
- **Constant Functions:** For the deque class I made print, size and is_empty constant functions. For the doubly linked list class I made front, back, search and is_empty constant. For the node class I made get_next, get_prev, get_key and get_value constant. This is because all of these functions should not change any of my member variables.
- **No Constant parameters or operator overloading used:** All of my function parameters are either strings, pointers or integers and so they are being passed by value meaning it wouldn't make much sense to make them constant. I also had no need to use operator overloading in this project.

Test Cases

Strategy:

A) Brainstorm all possible edge cases that may cause bugs or errors in my program
B) Handwrite solutions to all my possible edge cases and write what my expected output should be
C) Verify through my program that I get the same results as expected through the print function and debug/fix if there are any errors
D) Verify through valgrind that I have no memory leaks

Sample Test Cases:

1. Creating a deque class with max size 3 - Print success
2. Calling pop_back when the deque is empty - Print failure
3. Calling pop_front when the deque is empty - Print failure
4. Calling push_front when the deque is not at max size - Print success and the deque should now hold that URL at the front

5. Calling push_back when the deque is not at max size - Print success and the deque should now hold that URL at the back
6. Calling push_front when the deque is at max size - Print success and delete the URL that was originally at the end of the deque
7. Calling pop_back when the deque is non empty - Print success and the last URL should be removed
8. Calling pop_front when the deque is non empty - Print success and the first URL should be removed
9. Calling clear on an empty deque - Print success and deque should stay empty
10. Calling clear on non empty deque- Print success and deque should now be empty

Performance Considerations

- **Find**: In my implementation find has an asymptotic tight bound of theta(n) where n is the number of elements in the deque. My find function does a linear search through all the nodes starting from the head. It sets a temp node to the head and iterates while that temp node is not null or when the temp nodes key is equal to the key we are looking for. Each iteration it sets the temp node to the next node. After the loop finishes the temp node is returned. In the worst case the key does not exist in the deque and the while loop does n iterations. Checking the while loop condition is constant time and returning the temp node is also constant time so overall the tight bound running time of find is theta(n). This all also means that the upper bound is O(n) since the tight bound is also an upper bound.
- **Clear:** In my implementation clear has an asymptotic tight bound of theta(n) where n is the number of elements in the deque. Clear continuously calls the pop_back function while the deque is not empty. Each time pop_back is called the size is reduced by 1 so the number of times clear calls pop_back is n times. The pop_back function is done in constant time since it only consists of constant time operations. As a result the overall tight bound running time of clear is theta(n). This also means that the upper bound is O(n) since the tight bound also represents an upper bound.
- **Print:** In my implementation print has an asymptotic tight bound of theta(n) where n is the number of elements in the deque. My print function sets a temp node to the end of the list and traverses backwards using a while loop until it reaches nullptr (the prev node of the head). As a result the number of iterations the while loop does is n. Each iteration prints the current key and value of the temp node which takes constant time. Therefore the overall running time of print is theta(n). This also means the upper bound is O(n) since the tight bound also represents an upper bound.
- **Others:** All of my other functions only use constant time operations and therefore run in O(1). This is done by manipulating pointers to prevent needing to traverse through the entire linked list a lot.