ECE 250 - Project 1
Deque as Liked List
Design Document

Harshit Manchanda, UW UserID: h2mancha

Jan 31st, 2020

1. Overview of Classes

<u>**Class 1:** Node</u>

Description:
Represents a node of the Deque doubly linked list and stores the address of the previous and next nodes along with the integer value given to it.

Member Variables:
1. Node Value (Integer value to be added)
2. Node type pointer next (Pointing to the next node)
3. Node type pointer previous (Pointing to the previous node)

For example, a node in the deque list will hold the value 10 as the node value and the addresses of the next and previous node in the next and previous pointer respectively.

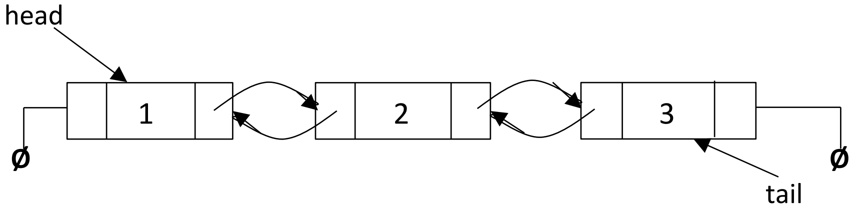
Member Functions (operations):
Doesn't have any member functions because it has Deque as its friend class so the private member variables can be directly accessed and used inside the Deque class.

<u>**Class 2:** Deque</u>

Description:
It is a queue which is implemented using a double linked list data structure, stores integer values in the nodes, which are objects of Node class, allows insertions and deletions at both ends and can dynamically grow in size. It has more functions like clearing all values, printing them, etc.

Member Variables:
1. Size of the deque list
2. Node type pointer head (Points to the head of the list)
3. Node type pointer tail (Points to the tail of the list)

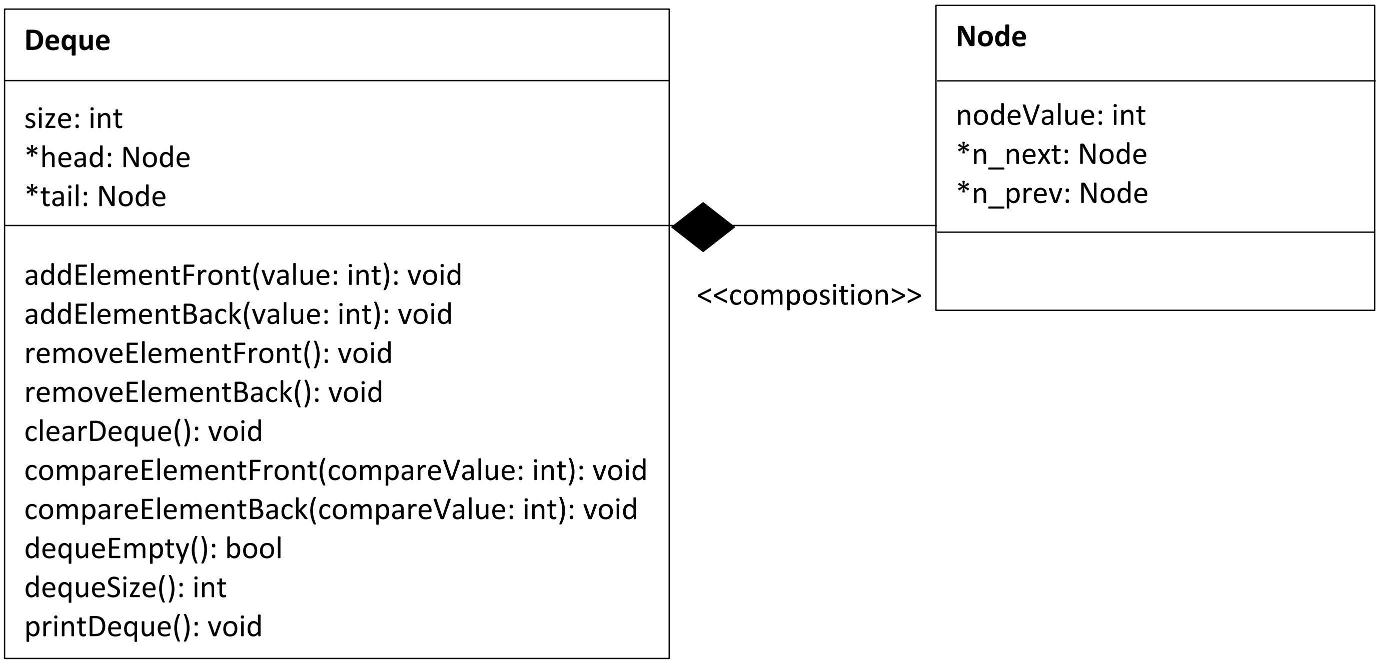
For example, the playlist for -
enqueue_front 2
enqueue_front 1
enqueue_back 3
will be represented as list:



Member Functions (operations):
1. <u>addElementFront:</u> adds a new node at the front of the linked list and becomes the new head. It stores the value passed as a parameter and also points to the next and previous nodes.
2. <u>addElementBack:</u> adds a new node at the back of the linked list and becomes the new tail. It stores the value passed as a parameter and also points to the next and previous nodes.

3. removeElementFront: removes the node at the front of the linked list by deallocating the memory and makes the head the adjacent node.
4. removeElementBack: removes the node at the back of the linked list by deallocating the memory and makes the tail the adjacent node.
5. clearDeque: It clears the whole deque by deleting all the nodes if it is not already empty.
6. compareElementFront: It compares the value passed as a parameter with the value stored in the node at the front(head) of the linked list.
7. compareElementBack: It compares the value passed as a parameter with the value stored in the node at the back(tail) of the linked list.
8. dequeEmpty: It checks if the deque is empty or not.
9. dequeSize: It returns the current size of the deque list
10. printDeque: prints the node values of list, first from front to back and then from back to front.

UML Class Diagram -


Deque
size: int
*head: Node
*tail: Node
addElementFront(value: int): void
addElementBack(value: int): void
removeElementFront(): void
removeElementBack(): void
clearDeque(): void
compareElementFront(compareValue: int): void
compareElementBack(compareValue: int): void
dequeEmpty(): bool
dequeSize(): int
printDeque(): void
<<composition>>
Node
nodeValue: int
*n_next: Node
*n_prev: Node

2. Constructors/Destructors

Class Node (Constructor):
The constructor for this class assigns the integer value, next and previous pointer addresses passed as parameters to the node value variable, next and previous pointers for that node respectively. There is no need for a default constructor because whenever a new node is created, a value is always passed with enqueue_front/back command.

Class Node (Destructor):
The destructor needs to free the memory used for creating the node and does so by deleting the next and previous pointers by making them as null pointers.

Class Deque (Constructor):
The default constructor takes in nothing and creates an empty linked list with the head and tail pointers being null pointers and size of the list as 0.

Class Deque (Destructor):
The destructor is necessary for this class since we have dynamic nodes and the memory isn't freed on its own at the end of the program so there is a need to manually deallocate the memory. For this, the destructor deletes the head and tail pointers by making them null.

3. Test Cases // To Do

Test 1: Create a new deque by adding nodes at the front and back, print them, then clear it and print it again to check if anything is printed or not.

Test 2: Create a few nodes in the deque list and then delete them one by one until the list is empty and then check if the empty function returns the correct result.

Test 3: Add some values in the list and then compare the front and back values if they match or not with the values in the node

Test 4: Check if the correct size is returned after adding the deleting nodes and also when it is cleared

Test File Example 1:

```
enqueue_back 1
enqueue_back 2
front 1
back 10
back 2
empty
print
clear
dequeue_front
dequeue_back
empty
enqueue_front 4
print
```

Test File Example 2:

```
dequeue_front
dequeue_back
front 0
back 0
clear
empty
enqueue_front 1
front 1
back 1
print
enqueue_front 2
enqueue_front 3
front 3
print
```

4. Performance

In the Deque class, besides the print and clear function, all the functions have single line statements, conditional statements which all together have the time complexity of O(1) which is constant time.

The print and clear functions both have while loops which traverse through the list from 1 to n(size of the list) which means it executes for every element in the list and that makes the time complexity of O(n).

References:

For test cases - https://github.com/ece23/ECE250-testCases
For help in code – Tutorial Linked List project