ECE250 – Project 1
Deque Data Structure - Design Document

Raymond Chen, UW UserID: r276chen Jan 29th,2020

Overview of Classes

Class:

Deque

Description:

Represent a data structure to store integer values using a doubly linked list. This data structure performs functions like delete, retrieve and add integers to the linked list.

Member variables:

- INT VAR: stores the size of the linked list
- POINTER VAR; points to the head of the linked list
- POINTER VAR: points to the tail of the linked list

Member functions:

- VOID enqueue_front – adds to the front of the linked list
- VOID enquene_back – adds to the back of the linked list
- VOID dequeue_back – removes from the back of the linked list
- VOID dequene_front – removes from the front of the linked list
- BOOL empty – checks to see if the linked list is empty
- VOID clear – remove all elements in the linked list
- VOID front – check to see if the front of the linked list is equal to given value
- VOID back – check to see if the back of the linked list is equal to the given value
- INT size – return the size of the linked list
- VOID print – prints the integers values in the linked list in forward and backward order

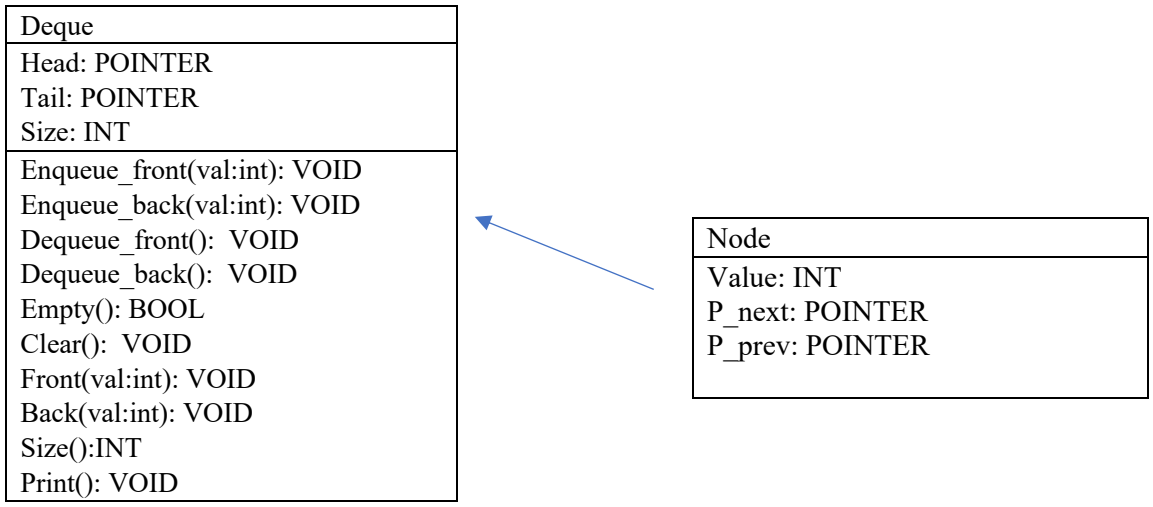
Class:

Node

Description:

Represent an integer to be added into the linked list **Member variables:**

- INT VAR: stores the integer value
- POINTER VAR: pointer that points to the next integer value
- POINTER VAR: pointer that points to the previous integer value



Constructors/Destructor/Operator overloading

Class: Deque

Deque () – initial size to 0 and all pointers to nullptr

~ Deque () – calls clear() to delete pointers

Class: Node

Node (INT&) – create a new node with assigned integer value and its pointer to nullptr

~ Node () – assign all pointers to nullptr to avoid dangling pointers

Copy constructors will not be need since we do not require a deep copy

Test Cases

Basic Cases:

- Inserting and deleting nodes from the front and back. Check for wild and dangling pointers
- Test print and clear to check for illegal memory locations
- Test to see if size is incremented or decremented on every add/delete
- Perform operations on a linked list with many items

Edge Cases:

- Perform operations on an empty linked list
- Perform operations on a linked list with one value
- Perform operations on a linked with duplicated values
- Test for non-integer inputs

Average Case		Edge Case	
Input	Output	Input	Output
enqueue_front 4	success	size	size is 0
enqueue_front 6	success	clear	success
enqueue_back 1	success	print	
enqueue_back 3	success	empty	success
enqueue_back 5	success	enqueue_front 1	success
size	size is 5	enqueue_front 1	success
print	6 4 1 3 5 5 3 1 4 6	enqueue_front 1	success
empty	failure	enqueue_front 1	success
clear	success	enqueue_front 1	success
enqueue_back 1	success	size	size is 5
print	1 1	empty	failure
size	size is 1	print	1 1 1 1 1 1 1 1 1 1
empty	failure	clear	success

Time Complexity

It is required to add and delete nodes in the linked list in linear time. This means traversing the linked list is not allowed. To do so, an access pointer will be required to access the element at the front of the linked list. Since nodes can also be deleted from the end of the linked list, another access pointer is required for the last element in the linked list.