Project 1 Design Document

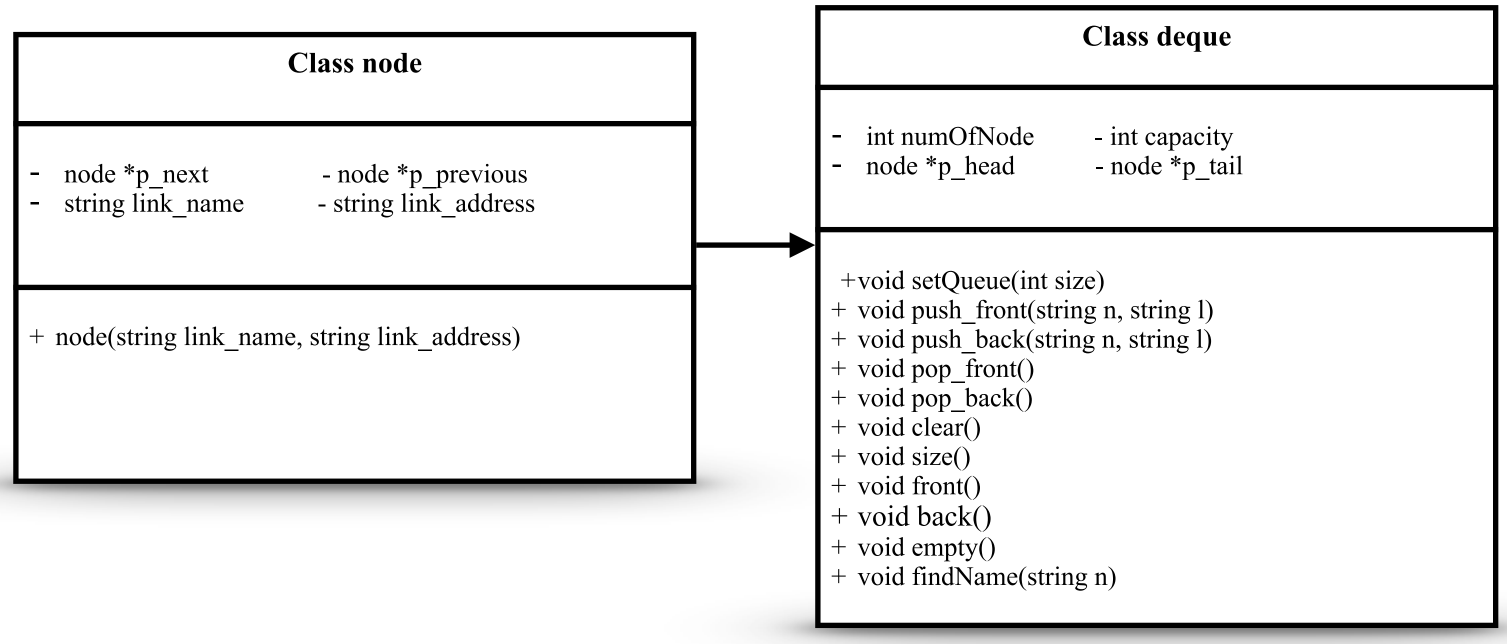
Bill Yao (b29yao)

1. Overview of classes

In this project, I have designed two classes to implement the functionality of the program.

The first class “node” stores the link name and link address of the input URL separately into two member string variables “link_name” and “link address”, as well as two pointers “p_next” and “p_previous” to link nodes internally together. Objects will be created based on this class to initialize, set values and get destroyed. The second class “deque” contains the major member functions as well as head and tail pointers to implement features in the linked list for each function. It also has two private member variables, “numOfNode” and “capacity” to detect the number of inputs in the linked list and reassure that it does not exceed the pre-set capacity. All the member function collaborates with a helper function “isFull()’ to make changes to the list.

2. UML class diagram



3. Design Decisions

For Class node, we need two pointers (p_next and p_previous) for each node to build up a connection with each other, and two member variables to separately store the data of the link name and link address from the input URL. We need to set up initial null values for each pointer and variable and delete the pointer in the destructor to prevent a memory leak. There are no operators that we used to override. When setting the link name and link address, we did not pass by reference on any parameters since they are constantly taking in new data as the input is being pushed in and popped out.

For Class deque, we need two member variables “numOfNode”, and “capacity” to check how many existing nodes are in the linked list, so that we can implement different steps in several functions based on the condition. Therefore, we created a helper function isFull(), to validate if the condition “full linked list” is met under each node pushing or popping. Also, we need p_head and p_tail to point to the head and tail of the linked list. All pointers are set to be null in the constructor and every node in the linked list needs to be cleared in the destructor, by traversing through the list and deleting each one of them.

4. Testing Strategy

When testing the entire program, we can divide it into multiple simple testing tasks that would internally back up each other. So we set up the test cases for each member function and make sure they all work properly, then we wrap them up for overall testing.

push_front()& push_back(): we can simply push several nodes into the queue under different scenarios (empty/with nodes/full), then use the print function to check with expected outcomes.

pop_front() & pop_back(): we pop out the head and tail nodes from the linked list and use the print function or front&back functions to prove the popping works.

clear() & size(): these two functions always come to testing together. After each pushing/popping, we can check if the size increases/decreases, and after clearing the linked list, we can also use the size function to make sure the list is completely cleared.

front() & back(): we start by pushing in some inputs into the list, and record the expected head and tail in the queue. Then we run front and back functions to validate the correctness.

empty() &findName(): after each clearing function, we call the empty function to validate it works, and after each pushing, we use findName() to make sure the input is in the queue.

print(): the print function should display all entries in the queue from the back of the list, and shouldn’t display anything if it is empty.

5. Performance considerations

Since this project is mostly constructed in a linked list, so the performance time is either O(n) (linear performance) or O(1).

O(n): for clear(), findName() and print(), these functions all need to traverse through the entire list and compute the output, and they all take a linear amount of time depending either on the input size or the size of the list.

O(1): for push_front(), push_back(), pop_front(), pop_back(), size(), empty(), front(), back(), these functions we either only use the member variable “numOfNode” and “capacity”, or we only access the head or tail of the linked list, so it only takes one unit time to perform the functionality.