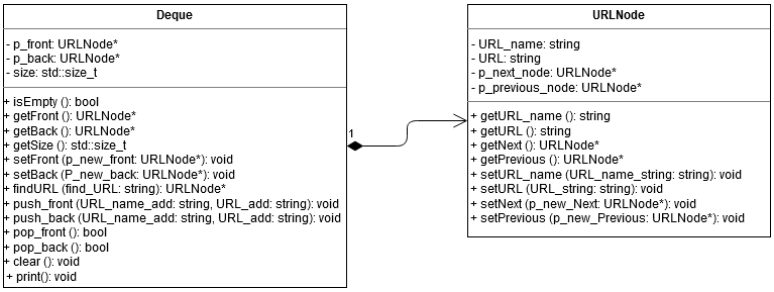
Design Document Project 1

Overview of Classes

Two classes were designed, a Deque class, and a URLNode class. The Deque class was designed to create and manage (add/remove/find/clear etc.) a double ended queue (linked list) containing a search history of URLs. The URLNode class was designed to contain information about individual URL's in the Deque (URL Name/URL) as well as information about the URL following/preceding it in the Deque (ie. pointers to the next/previous URLNode).

The relationship between the two classes is that the Deque class uses the URLNode objects as items in the Deque which store the URL information for each URL in the Deque. The Deque class can then manipulate the information in the Deque using the public member functions from the URLNode class.

UML Class Diagram



Design Decisions

Deque

The constructor uses for this class simply initializes the private member variables to ensure all the pointers start pointing to nullptr and that the initial size of the Deque is set to 0.

The destructor deallocates the memory for the Deque which is allocated in the push front/back functions by calling the clear () function. The clear () function calls pop_front() until the Deque is empty. The pop_front() function deallocates the memory (allocated in push front/back) for a single node reassigns to pointers appropriately. I used the pop_front() function in a loop in order to create the clear () function to remove all items from the Deque (doubles as the destructor). Reusing the pop_front() function to clear the whole Deque was designed to prevent memory leaks and dangling pointers from occurring. No operators were overwritten.

For the following member functions:

void setFront(URLNode const p_new_front); void setBack(URLNode* const p_new_back);*
URLNode findURL(const string find_URL); void push_front(const string URL_add_name, const string URL_add);*
push_back(const string URL_add_name, const string URL_add);

The keyword *const* was used at the parameter level since in each case the parameter being passed should not be altered or reassigned within the function.

For the following member functions:

bool isEmpty() const; URLNode getFront() const; URLNode* getBack() const; std::size_t getSize() const;*
URLNode findURL(const string find_URL) const;*

The keyword *const* was used at the function level since in each case the function should not change the value of any member variable.

For the following member functions:

bool pop_front(); bool pop_back(); void clear();

The keyword *const* was **not** used at all since in each case the function needs to be able to **change** the value of any member variable and has no parameters.

URLNode

The constructor for the URLNode class assigns an empty string to both *URL_name* and *URL* and initiates the pointers for the next and previous node to *nullptr* to ensure that the attributes of each object are empty/nullptr. No destructor was required since there was no dynamic memory allocation in the constructor. No operators were overridden.

For the following member functions:

void setURL_name(const string URL_name_string); void setURL(const string URL_string);
void setNext(URLNode const p_new_Next); void setPrevious(URLNode* const p_new_Previous);*

The keyword *const* was used at the parameter level since in each case the parameter being passed should not be altered or reassigned within the function.

For the following member functions:

string getURL_name() const; string getURL() const; URLNode getNext() const; URLNode* getPrevious() const ;*

The keyword *const* was used at the function level since in each case the function should not change the value of any member variable.

*Note there were no functions which had zero const at both the function and parameter level in URLNode class.

Test Cases

To test if my program met the specified requirements, I created specific inputs to try and cover the many possible cases which would reveal problems with my code (stack dump, memory leak, dangling pointer, etc.)

Some specific cases which I tested for:

- Back/Front of the deque when it is empty.
- Clear/Print/Size/Empty of the dequeue when it is empty.
- Popping/Pushing the front/back of the dequeue when it is empty.
- Popping the deque when there is only one URL
- Finding an item in the deque using different casing (find “google” – item in list “Google”)
- Finding an item which does not appear in the deque.
- Front/Back with only one item in the deque.
- Pushing/Finding an item in the deque which has a space “ “ in the URL Name

Performance Considerations

Expected run time of O(n)

Print, Clear and Find are all operations which must iterate through the entire deque. Clear and Print need to access every element in the Deque and in the worst case Find would also have to iterate through all elements in the Deque (item is located at the end of the Deque). Hence the asymptotic upper bound is O(n) since in the worst case the operation would have to go through all n elements in the Deque.

Expected run time of O(1)

All other operations (exit, empty, back, front, size, pop_back, pop_front, push_back, push_front) have an expected running time of O(1) because they only require the reassignment of variables, checking of conditions, allocation of memory, and the accessing/changing of member variables. For example, size can be computed in a constant time since it is stored as a member variable and updated (incremented/decremented) whenever an item is pushed/popped from the Deque. This design choice allows size to be computed in a constant time instead of O(n) which would be the case if the deque was iterated through each time to calculate the size.