Code Reflection and Pseudocode

Jorgo Qendro

Southern New Hampshire University

CS-300: Analysis and Design

Professor Adamo

September 21st,2024

Code Reflection

The implementation of this singly linked list in C++ follows a straightforward approach, focusing on the basic operations of appending, prepending, printing, searching, and removing nodes from the list. The list is designed to manage bids for property auctions, where each bid is stored in a “Node” structure. The “LinkedList” class manages a dynamic list, starting with a head and tail pointer to track the first and last nodes, respectively. The “Append” function adds a new bid to the end of the list. If the list is empty, the new bid becomes both the head and the tail. Otherwise, the current tail's next pointer is updated to point to the new node, and the tail itself is updated to this new node. The size of the list is incremented.

The “Prepend” function adds a new bid to the beginning of the list. If the list is empty, the new node becomes both the head and the tail, like the “Append” function. If the list already contains nodes, the new node's next pointer is set to the current head, and the head is updated to this new node. The size of the list has also increased. The “PrintList” function iterates through the list from the head to the tail, outputting each bid's details to the console. It continues until it reaches the end of the list, where the current pointer becomes null.

The “Remove” function allows for the deletion of a node by its “bidId”. Special care is taken if the node to be removed is the head, in which case the head is updated to point to the next node. If the node is not the head, the function iterates through the list, adjusting the next pointers to remove the desired node from the chain, freeing up its memory and reducing the size of the list. The “Search” function traverses the list to find a node by its “bidId". If a matching node is found, it returns the bid associated with that node; if not, it returns an empty bid. Finally, the “Size” function simply returns the current number of nodes in the list.

In conclusion, the overall structure of the “LinkedList” class ensures that common list operations can be performed efficiently. The code uses proper memory management techniques, such as freeing nodes in the destructor and carefully handling memory during the remove operation, to avoid memory leaks.

**Pseudocode**

START

Procedure Append(bid):

Create a new node with the given bid

IF list is empty THEN:

Set the new node as both head and tail

ELSE:

Set tail's next to the new node

Update the tail to the new node

END IF

Increment the size of the list

END PROCEDURE

Procedure Prepend(bid):

Create a new node with the given bid

IF list is not empty THEN:

Set new node's next to the current head

END IF

Set new node as the new head

IF list was empty THEN:

Set tail to the new head

END IF

Increment the size of the list

END PROCEDURE

Procedure PrintList:

Set current to head

WHILE current is not null DO:

Display current bid's details

Set current = current.next

END WHILE

END PROCEDURE

Procedure Remove(bidId):

IF head is null THEN:

RETURN (Nothing to remove)

END IF

IF head's bidId matches bidId THEN:

Set head = head.next

Decrement size

RETURN

END IF

Set current to head

WHILE current.next is not null DO:

IF current.next.bidId matches bidId THEN:

Set current.next = current.next.next

Decrement size

RETURN

END IF

Set current = current.next

END WHILE

END PROCEDURE

Procedure Search(bidId):

Set current to head

WHILE current is not null DO:

IF current bidId matches bidId THEN:

RETURN current bid

END IF

Set current = current.next

END WHILE

RETURN empty bid (not found)

END PROCEDURE

Procedure Size:

RETURN size (the number of nodes in the list)

END PROCEDURE

END