```cpp
/**
 * @file -  LinkedListStack.cpp
 *
 * Implements StackADT<T> using a linked list as the internal data structure for the stack elements.
 *
 * @author - Humayun Kabir, Instructor, CSCI 161, VIU
 * @version - 0.0.1
 * @date - April 25, 2021
 *
 */



#include <iostream>
#include "StackADT.h"

using namespace std;

template <typename T>
class LinkedListStack: public StackADT<T> {

private:

  /*
   * Node<T>
   * Private Inner class for LinkedListStack<T> class.
   * This class will be used only inside the LinkedListStack<T> class.
   * For that reason, its member variable data is not required to be encapsulated (private).
   */
  class Node {
      public:
        T data;
            Node* next;
        Node(): next(NULL) {}
        Node(T data): data(data), next(NULL) {}
        ~Node() {}
  };

  /*
   * @brief - Copies all the linked nodes from 'src' to 'dst'
   *
   * Internal helper function for LinkedListStack<T> class to copy and link
   * all the linked nodes starting at the node 'src'.
   *
   * @param - dst - the first node of the copied and linked nodes.
   * @param - src - the first node of linked nodes that is being copied.
   */
  void deepCopy(Node*& dst, const Node* src) {
    if(src == NULL) {
        dst = NULL;
        return;
    }
    dst = new Node(src->data);
        Node* copy = dst;
    Node* next = src->next;
    while(next != NULL) {
        copy->next = new Node(next->data);
        copy = copy->next;
        next = next->next;
    }
  }

  /*
   * @brief - Deletes all the linked nodes starting from 'node'.
   *
   * Internal helper function for LinkedListStack<T> class to delete all the linked nodes starting at 'node'.
```

```
67       *
68       * @param - node - the first node of the linked nodes that are being deleted.
69       */
70      void deepClean(Node*& node) {
71        while (node != NULL) {
72            Node* remove = node;
73            node = node->next;
74            delete remove;
75        }
76      }
77
78      int size;
79      Node* top;
80
81    public:
82
83      /*
84       * Default constructor
85       */
86      LinkedListStack(): StackADT<T>::StackADT(), size(0), top(NULL) {
87        cout<<"LinkedListStack::default constructor......"<<endl;
88      }
89
90      /*
91       * Copy constructor
92       */
93      LinkedListStack(const LinkedListStack& copy): StackADT<T>::StackADT(), size(copy.size),
         top(NULL) {
94        deepCopy(top, copy.top);
95        cout<<"LinkedListStack::copy constructor......"<<endl;
96      }
97
98      /*
99       * Move constructor
100      */
101     LinkedListStack(LinkedListStack&& temp): StackADT<T>::StackADT(), size(temp.size), top(
        temp.top) {
102         temp.top = NULL;
103         cout<<"LinkedListStack::move constructor......"<<endl;
104     }
105
106     /*
107      * Destructor
108      */
109     ~LinkedListStack() {
110       deepClean(top);
111       cout<<"LinkedListStack::destructor......."<<endl;
112     }
113
114     /*
115      * Copy assignment operator
116      */
117     LinkedListStack& operator = (const LinkedListStack& copy) {
118       if( this == &copy) {
119           return *this;
120       }
121       deepClean(top);
122       deepCopy(top, copy.top);
123       size = copy.size;
124       cout<<"LinkedListStack::copy assignment......"<<endl;
125       return *this;
126     }
127
128     /*
129      * Move assignment operator
130      */
131     LinkedListStack& operator = (LinkedListStack&& temp) {
132         if( this == &temp) {
133             return *this;
```

```
134         }
135         deepClean(top);
136         top = temp.top;
137         size = temp.size;
138         temp.top = NULL;
139         cout<<"LinkedListStack::move assignment......"<<endl;
140         return *this;
141     }
142 
143 
144     /**
145      * @brief - Pushes the element on the top of the stack and advances the top
146      *
147      * This function will be available to use with LinkedListStack<T> object and both
     LinkedListStack<T> and StackADT<T>
148      * references.
149      *
150      * @param - element, the element to be pushed onto the stack.
151      */
152     void push(T element) override {
153       Node* node = new Node(element);
154       node->next = top;
155       top = node;
156       size++;
157     }
158 
159 
160      /**
161      * @brief - Pops the top element of the stack and reverts the top
162      *
163      * This function will be available to use with LinkedListStack<T> object and both
     LinkedListStack<T> and StackADT<T>
164      * references.
165      *
166      * @return - top element
167      */
168     T pop() override {
169       if(top != NULL) {
170         Node* node = top;
171         T data = node->data;
172         top = node->next;
173         delete node;
174         size--;
175         return data;
176       }
177       else {
178         throw "Stack is Empty!";
179       }
180     }
181 
182 
183      /**
184      * @brief - Gives the top element of the stack and does not revert the top
185      *
186      * This function will be available to use with LinkedListStack<T> object and both
     LinkedListStack<T> and StackADT<T>
187      * references.
188      *
189      * @return - top element
190      */
191     T peek() override {
192       if( top != NULL) {
193         return top->data;
194       }
195       else {
196         throw "Stack is Empty";
197       }
198     }
199 
```

```
200
201      /**
202       * @brief - Gives the current size or the number of elements that has been pushed inot
          the stack but not popped yet.
203       *
204       * This function will be available to use only with LinkedListStack<T> object or
          reference but not with a
205       * StackADT<T> reference.
206       *
207       * @return - current size of the stack
208       */
209      int getSize() override {
210        return size;
211      }
212
213
214      /**
215       * @brief - Returns true if the stack has no element to pop, false otherwise.
216       *
217       * This function will be available to use only with LinkedListStack<T> object or
          reference but not with a
218       * StackADT<T> reference.
219       *
220       * @return - true if the stack is empty.
221       */
222      bool isEmpty() override {
223        return size == 0 || top == NULL;
224      }
225
226    };
227
```