UNIVERSIDAD NACIONAL DE SAN AGUSTÍN DE AREQUIPA

FACULTAD DE INGENIERIA DE PRODUCCION Y SERVICIOS

ESCUELA PROFESIONAL DE CIENCIAS DE LA COMPUTACIÓN

LABORATORIO 10 – Templates – Sobrecarga de Operadores.

DOCENTE:

Enzo Edir Velásquez Lobatón

ALUMNO:

Owen Haziel Roque Sosa.

FECHA:

22/05/2022

Arequipa – Perú

```cpp
#ifndef LISTA_H
#define LISTA_H
#include <iostream>
using namespace std;

template <class T>
class Lista;

template <class T>
class Nodo {
public:
    Nodo<T>(T, Nodo* =nullptr);
    friend class Lista<T>;
private:
    T value;
    Nodo* sig;
};

template <class T>
Nodo<T>::Nodo(T _value, Nodo* _sig) {
    sig = _sig;
    value = _value;
}
```

1. *Clase Nodo*

```cpp
template <class T>
class Lista {
public:
    Lista<T>();
    ~Lista<T>();
    void insertarInicio(T);
    void insertarFinal(T);
    void insertar(T,unsigned int);
    void eliminarInicio();
    void eliminarFinal();
    void eliminar(unsigned int);
    void imprimir();
    void imprimir(Lista<T>*);
    void ordenamiento();
    unsigned int getSize();

private:
    Nodo<T> * first = nullptr;
    unsigned int size = 0;

};

//constructor
template <class T>
Lista<T>::Lista(){}
```

2. *Clase Lista*

```cpp
//destructor
template <class T>
Lista<T>::~Lista() {
    Nodo<T> *aux = first;
    Nodo<T> *delete_node;
    while(aux != nullptr) {
        delete_node = aux;
        aux = aux->sig;
        delete delete_node;
    }
    first = nullptr;
    size = 0;
}
```

3. *Clase Lista - Destructor*

```cpp
// mostrar lista
template <class T>
void Lista<T>::imprimir(){
    Nodo<T> *aux = first;
    while(aux != nullptr) {
        cout << aux->value << " -> ";
        aux = aux->sig;
    }
    cout << endl;
}
```

4. *Imprimir Lista*

1. Defina una lista enlazada que permita insertar elementos al final de todos los elementos que ya se hayan ingresado. Por el momento no es necesario preservar un orden, simplemente los elementos nuevos deben de ingresar como el último elemento.

```
//ej 1
template <class T>
    void Lista<T>::insertarFinal(T val){
    Nodo<T> *nuevo = new Nodo<T>(val);
    nuevo->sig = NULL;
    if (first == NULL) { // si es el primer elemento
        first = nuevo;  // a añadir a la lista
    }
    else {
        Nodo<T> * aux = first;
        while(aux->sig!=nullptr){ // recorre actualizando
            aux = aux->sig; // el valor de aux hasta llegar
        } // al ultimo valor, que apunta a nullptr por defecto
        aux->sig = nuevo;
    }
    size++;
}
```

```
#include<iostream>
#include"Lista.h"
using namespace std;

int main (int argc, char *argv[])
    Lista<int> A;
    A.insertarFinal(450);
    A.insertarFinal(8);
    A.insertarFinal(10);
    A.insertarFinal(51);
    A.imprimir();
    return 0;
}
```

```
450 -> 8 -> 10 -> 51 ->
```

2. Con la implementación de la lista enlazada anterior, desarrolle una función que permita ingresar los elementos al inicio de todos los demás elementos. Tendrá que modificar el comportamiento del puntero que tiene referencia al primer elemento para que sea redireccionado al nuevo elemento por ingresar.

```
//ej 2
template <class T>
void Lista<T>::insertarInicio(T val){
    Nodo<T> *nuevo = new Nodo<T>(val);
    if (first == nullptr) { // sí es el primer elemento
        nuevo->sig = nullptr; // a añadir a la lista
        first = nuevo;
    }
    else { // sí hay elementos en la lista, agrega
        nuevo->sig = first; //en primera posicion (first)
        first = nuevo;
    }
    size++;
}
```

```
#include<iostream>
#include"Lista.h"
using namespace std;

int main (int argc, char *argv[]) {
    Lista<int> A;
    A.insertarFinal(450);
    A.insertarFinal(8);
    A.insertarFinal(10);
    A.insertarFinal(51);
    A.insertarInicio(4);
    A.insertarInicio(75);
    A.imprimir();
    return 0;
}
```

```
75 -> 4 -> 450 -> 8 -> 10 -> 51 ->
```

3. Desarrolle una función que permita ingresar elementos en el medio de dos elementos de la lista enlazada, como se muestra en la siguiente imagen. Solicite que se ingrese una posición válida dentro de la lista y permita que el valor ingresado se pueda anexar en esa posición.

```
//ej 3
template <class T>
void Lista<T>::insertar(T val, unsigned int idx){
    //validar
    if (idx == 0){
        Lista<T>::insertarInicio(val);
        return;
    }
    if (idx == size) {
        Lista<T>::insertarFinal(val);
        return;
    }
    else if (idx > size){
        cout << "Indice fuera de rango.\n";
        return;
    }
    Nodo<T> * aux = first;
    for (unsigned int i = 0; i < idx-1 ; i++){ //recorremos
        aux=aux->sig; // hasta llegar al idx-1 indicado
    }
    Nodo<T> *nuevo = new Nodo<T>(val,aux->sig);
    aux->sig = nuevo;
    size++;
}
```

```
#include<iostream>
#include"Lista.h"
using namespace std;

int main (int argc, char *argv[]) {
    Lista<int> A;
    A.insertarFinal(450);
    A.insertarFinal(8);
    A.insertarFinal(10);
    A.insertarFinal(51);
    A.insertarInicio(4);
    A.insertarInicio(75);
    A.imprimir();
    A.insertar(14,5);
    A.imprimir();
    A.insertar(78,3);
    A.imprimir();
    return 0;
}
```

```
75 -> 4 -> 450 -> 8 -> 10 -> 51 ->
75 -> 4 -> 450 -> 8 -> 10 -> 14 -> 51 ->
75 -> 4 -> 450 -> 78 -> 8 -> 10 -> 14 -> 51 ->
```

4. Elabore una función que permita eliminar el último elemento de una lista enlazada. (Evite copiar los elementos en una nueva lista para completar la eliminación del elemento).

```cpp
// ej4
template <class T>
void Lista<T>::eliminarFinal(){
    if (first == nullptr) {
        cout << "No existen elementos en la lista.\n";
        return;
    } // si es el primer elemento a añadir a la lista
    else {
        Nodo<T> * delete_node = first; // asigno delete a first
        for(unsigned int idx = 0; idx < size-2; idx++){ // recorre
            delete_node = delete_node->sig; // hasta el penultimo
        }
        delete_node->sig = nullptr; // establece que es el ultimo, apunta a nada
        delete delete_node->sig;
    }
    size--;
}
```

```cpp
#include<iostream>
#include"Lista.h"
using namespace std;

int main (int argc, char *argv[]) {
    Lista<int> A;
    A.insertarFinal(450);
    A.insertarFinal(8);
    A.insertarFinal(10);
    A.insertarFinal(51);
    A.insertarInicio(4);
    A.insertarInicio(75);
    A.insertar(14,5);
    A.insertar(78,3);
    A.imprimir();
    A.eliminarFinal();
    A.imprimir();
    return 0;
}
```

```
75 -> 4 -> 450 -> 78 -> 8 -> 10 -> 14 -> 51 ->
75 -> 4 -> 450 -> 78 -> 8 -> 10 -> 14 ->
```

5. Desarrolle una función que permita eliminar el primer elemento de una lista sin perder referencia de los demás elementos que ya se encuentran almacenados en la estructura. (Evite copiar los elementos en una nueva lista para completar la eliminación de los elementos)

```
//ej 5
template <class T>
void Lista<T>::eliminarInicio(){
    if (first == nullptr) {
        cout << "No existen elementos en la lista.\n";
        return;
    } // si es el primer elemento a añadir a la lista
    else {
        Nodo<T> * delete_node = first; // asigno delete a first
        first = first->sig; // first recorre al siguiente
        delete delete_node;
    }
    size--;
}
```

```
#include<iostream>
#include"Lista.h"
using namespace std;

int main (int argc, char *argv[]) {
    Lista<int> A;
    A.insertarFinal(450);
    A.insertarFinal(8);
    A.insertarFinal(10);
    A.insertarFinal(51);
    A.insertarInicio(4);
    A.insertarInicio(75);
    A.insertar(14,5);
    A.insertar(78,3);
    A.imprimir();
    A.eliminarInicio();
    A.imprimir();
    return 0;
}
```

```
75 -> 4 -> 450 -> 78 -> 8 -> 10 -> 14 -> 51 ->
4 -> 450 -> 78 -> 8 -> 10 -> 14 -> 51 ->
```

6. Dado una posición válida dentro de la lista, permita al usuario eliminar un elemento de cualquier posición sin perder referencia de los demás elementos.

```cpp
// ej 6
template <class T>
void Lista<T>::eliminar(unsigned int idx){
    //validar
    if (idx == 0){
        Lista<T>::eliminarInicio();
        return;
    }
    if (idx == size) {
        Lista<T>::eliminarFinal();
        return;
    }
    else if (idx > size){
        cout << "Indice fuera de rango.\n";
        return;
    }
    Nodo<T> * prev_node = first;
    //recorremos hasta llegar al idx-1 indicado
    //prev_node = nodo anterior al eliminar
    for (unsigned int i = 0; i < idx-1 ; i++){
        prev_node=prev_node->sig;
    }
    //nodo siguiente al eliminar
    Nodo<T> * next_node = prev_node->sig->sig;
    //elimino el nodo
    delete prev_node->sig;
    // asigno el nuevo siguiente
    prev_node->sig = next_node;
    size--;
}
```

```cpp
#include<iostream>
#include"Lista.h"
using namespace std;

int main (int argc, char *argv[]) {
    Lista<int> A;
    A.insertarFinal(450);
    A.insertarFinal(8);
    A.insertarFinal(10);
    A.insertarFinal(51);
    A.insertarInicio(4);
    A.insertarInicio(75);
    A.insertar(14,5);
    A.insertar(78,3);
    A.imprimir();
    A.eliminar(2);
    A.imprimir();
    return 0;
}
```

```
75 -> 4 -> 450 -> 78 -> 8 -> 10 -> 14 -> 51 ->
75 -> 4 -> 78 -> 8 -> 10 -> 14 -> 51 ->
```

7. Desarrolle un algoritmo de ordenamiento que permita ordenar los elementos de forma ascendente y descendente (8.) de acuerdo a la elección del usuario. Se debe poder simular el ingreso de 10 mil elementos de forma aleatoria y ordenarlos en el menor tiempo posible (< 2 seg).

```
template <class T>
void Lista<T>::ordenamiento(){
    // lista copia a ordenar
    Lista<T>* ord_list = new Lista<T>;
    Nodo<T> * aux = first;
    for (unsigned int idx = 0; idx < size; idx++){
        ord_list->insertar(aux->value,idx);
        aux = aux->sig;
    }
    Nodo<T> * act = ord_list->first;
    // nodo actual = head/primero
    T auxi;
    char opt;
    cout << "Ascendente o Descendente\t a|d: ";
    cin >> opt;
    switch (opt) {
    case 'a':
        while (act->sig != nullptr){
            Nodo<T> * next = act->sig;
            while (next!=nullptr){
                if(act->value > next->value){
                    //switch valores
                    auxi = next->value;
                    next->value = act->value;
                    act->value=auxi;
                }
                next=next->sig; // actualizar el valor de next
            } // ya que while no lo hará
            // recorrer actual y siguiente
            act = act->sig;
            next = act->sig;
        }
        Lista<T>::imprimir(ord_list);
        break;
```

```
#include<iostream>
#include"Lista.h"
using namespace std;

int main (int argc, char *argv[]) {
    Lista<int> A;
    A.insertarFinal(450);
    A.insertarFinal(8);
    A.insertarFinal(10);
    A.insertarFinal(51);
    A.insertarInicio(4);
    A.insertarInicio(75);
    A.insertar(14,5);
    A.insertar(78,3);
    A.imprimir();
    A.eliminar(2);
    A.ordenamiento();
    return 0;
}
```

```
75 -> 4 -> 450 -> 78 -> 8 -> 10 -> 14 -> 51 ->
Ascendente o Descendente          a|d: a
4 -> 8 -> 10 -> 14 -> 51 -> 75 -> 78 ->
```

8. Algoritmo de ordenamiento de forma descendente:

```
case 'd':
    while (act->sig != nullptr){
        Nodo<T> * next = act->sig;
        while (next!=nullptr){
            if(act->value < next->value){ //switch valores
                auxi = next->value;
                next->value = act->value;
                act->value=auxi;
            }
            next=next->sig; // actualizar el valor de next
        } // ya que while no lo hará
        // recorrer actual y siguiente
        act = act->sig;
        next = act->sig;
    }
    Lista<T>::imprimir(ord_list);
    break;
default:
    cout<<"Char No valido.\n";
    ord_list->~Lista();
}
```

```
75 -> 4 -> 450 -> 78 -> 8 -> 10 -> 14 -> 51 ->
Ascendente o Descendente          a|d: d
78 -> 75 -> 51 -> 14 -> 10 -> 8 -> 4 ->
```

- Método para imprimir lista a través de parámetro (Puntero a objeto Lista<T>)

```
template <class T>
void Lista<T>::imprimir(Lista<T>* ord_list){
    Nodo<T> *aux = ord_list->first;
    while(aux != nullptr) {
        cout << aux->value << " -> ";
        aux = aux->sig;
    }
    cout << endl;
    ord_list->~Lista(); // eliminar lista ordenada
}
```