

Codigo

```

#ifndef SLLIST_H
#define SLLIST_H
#include <iostream>
#include <utility>
template <typename Object> //Declaramos que se va a utilizar un template y los
object seran remplazados por un clase o tipo de dato.
class SList { //Aqui es la clase.
private: //funciones privadas
struct Node {
Object data;
Node *next;
//aqui hacemos el constructor tipo default
Node(const Object &d = Object{}, Node *n = nullptr) //aqui hacemos el
constructor de tipo copia
: data{d}, next{n} {} //Apunta a n, con valores de n
Node(Object &&d, Node *n = nullptr) //Declara un constructor de tipo
referencia constante.
: data{std::move(d)}, next{n} {} //Esta moviendo datos de donde se
originan hacia move, y con n es igual.
};
public:
class iterator { //Aqui empezamos a hacer la clase de iterador
public: //Aqui va todo lo publico
iterator() : current{nullptr} {} //Aqui se hacemos el constructor defeault
Object &operator*() { //Aqui se declaramos el puntero
if(current == nullptr) //Si el puntero es null, se dara mensaje de

```

Documentacion Lista.txt 2023-10-06

2 / 4

```

error
throw std::logic_error("Trying to dereference a null pointer.");
return current->data; //Rertorna los datos
}
iterator &operator++() { //Declaramos que vaya a la siguiente posicion del
iterador
if(current) //Si el puntero esta en la posicion actual, vaya a la
siguiente
current = current->next;
else //Si el puntero no tiene next, se dara mensaje de error
throw std::logic_error("Trying to increment past the end.");
return *this;
}
iterator operator++(int) { //aqui determinamos donde se usa el ++ por su
posicion
iterator old = *this;
++(*this);
return old;
}

```

```

iterator &operator--() { //Declaramos que vaya a la siguiente posicion del
iterador
    if(current) //Si el puntero esta en la posicion actual, vaya al
anterior
        current = current->previous;
    else
        throw std::logic_error("Trying to increment past the end.");
    return *this;
}

iterator operator--(int) { //aqui determinamos donde se usa el -- por su
posicion
    iterator old = *this;
    --(*this);
    return old;
}

```

```

bool operator==(const iterator &rhs) const { //Si es verdadero en los dos
casos, retorna a true, si no, a false
return current == rhs.current;
}
bool operator!=(const iterator &rhs) const { //Declara != si los dos casos
son iguales, entonces es false, si no, es true
return !(*this == rhs);
}

```

private:

Node *current; //Aqui es un puntero

iterator(Node *p) : current{p} {} //Aqui es un constructor que puede hacer
iteradores

friend class SLList<Object>; //Declara un amigo para class SLList

};

public:

SLList() : head(new Node()), tail(new Node()), theSize(0) { //aqui hacemos el
constructor para la lista

head->next = tail;

}

~SLList() { //Aqui hacemos el destructor

clear();

delete head;

delete tail;

}

iterator begin() { return {head->next}; } //la usamos para obtener el inicio
de la lista

iterator end() { return {tail}; } //aqui la usamos el final de la lista

Documentacion Lista.txt 2023-10-06

3 / 4

int size() const { return theSize; } //Aqui definimos el tamaño de la lista

bool empty() const { return size() == 0; } //Es para comprobar si la lista
esta vacia, si la lista esta vacia, retorna a true, y viceversa

void clear() { while (!empty()) pop_front(); } //Aqui se usa para limpiar toda
la lista, solo funciona si no esta vacia

```

Object &front() { //Aqui damos el dato inicial, si es que lo hay
if(empty()) //Aqui da empty si la lista lo es, si no, da el valor
throw std::logic_error("List is empty.");
return *begin();
}

void push_front(const Object &x) { insert(begin(), x); } //Se utiliza para
poner un objeto nuevo a la lista por copia
void push_front(Object &&x) { insert(begin(), std::move(x)); } //aqui es por
referencia

void push_back(const Object &x) { insert(end(), x); } //Se utiliza para poner
un objeto anterior a la lista por copia
void push_back(Object &&x) { insert(end(), std::move(x)); } //aqui es por
referencia

void pop_front() { //lo utilizamos para borrar el elemento inicial, solo si no
esta vacio vacio
if(empty())
throw std::logic_error("List is empty.");
erase(begin());
}

iterator insert(iterator itr, const Object &x) { //Inserta el elemento donde
esta apuntado el puntero, por copia
Node *p = itr.current;
head->next = new Node{x, head->next};
theSize++;
return iterator(head->next);
}

iterator insert(iterator itr, Object &&x) { //este es por referencia
Node *p = itr.current;
head->next = new Node{std::move(x), head->next};
theSize++;
return iterator(head->next);
}

void insert(int entero, const Object &x)
{
    insert(Get_iterator(entero), x);
}

iterator Get_iterator(int a)
{
    iterator it = begin();
    for(int x = 0; x != a; x++){
        it++;
    }
    return it;
}

iterator erase(iterator itr) { //Aqui es donde en vez de insertarlo, lo borra
if (itr == end())
throw std::logic_error("Cannot erase at end iterator");
Node *p = head;

```

```
while (p->next != itr.current) p = p->next;
p->next = itr.current->next;
delete itr.current;
theSize--;
return iterator(p->next);
}

void erase(int borrar){
    erase(Get_iterator(borrar));
}

void printList() { //Aqui se imprime la lista obteniendo la posicion del
iterador, y solo lo repite hasta que la lista se acaba
```

Documentacion Lista.txt 2023-10-06

4 / 4

```
iterator itr = begin();
while (itr != end()) {
    std::cout << *itr << " ";
    ++itr;
}
std::cout << std::endl;
}

private:
Node *head; //Inicio de lista
Node *tail; //Terminar de la lista
int theSize;
void init() { //Inicializacion de variables
theSize = 0;
head->next = tail;
}
};
#endif
```