Documentacion

SLList

```
#ifndef SLLLIST_H
#define SLLLIST_H
#include <iostream>
#include <utility>
template <typename Object> //Declaramos que se va a ultilizar un template y los
object seran remplazados por un clase o tipo de dato.
class SLList { //Aqui es la clase.
private: //funciones privadas
struct Node {
Object data;
Node *next;
//aqui hacemos el contructor 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 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;
}
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

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 ista, 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 ultiliza 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 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);
}
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 printList() { //Aqui se imprime la lista obteniendo la posicion del iterador,
y solo lo repite hasta que la lista se acaba

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
```

DLList

```
#ifndef DLLIST_DLLIST_H
#define DLLIST_DLLIST_H

#include <iostream>
#include <utility>

template <typename Object>
class DLList{
private:
    struct Node  {
        Object data;
        Node *next;
        Node *prev;

        //Constructor de copia
        Node(const Object &d = Object{}, Node *n = nullptr, Node *p = nullptr)
```

```
                : data{d}, next{n}, prev{p} {}

        //Constructor de referncia
        Node(Object &&d, Node *n = nullptr, Node *p = nullptr)
                : data{std::move(d)}, next{n}, prev{p} {}
    };

public:
    class iterator{
    public:
        //constructor implicito, se hace nulo el puntero
        iterator() : current{nullptr} {}

        //Operador * para que se compporte como puntero
        Object &operator*() {
            if(current == nullptr)
                throw std::logic_error("Trying to dereference a null pointer.");
            return current->data;
        }

        //Movimiento
        //Operador hace que pueda moverse por la lista.
        iterator &operator++() {
            if(current)
                current = current->next;
            else
                throw std::logic_error("Trying to increment past the end.");
            return *this;
        }

        iterator operator++(int) {
            iterator old = *this;
            ++(*this);
            return old;
        }

        iterator &operator--() {
            if(current)
                current = current->prev;
            else
                throw std::logic_error("Trying to decrease past the beginning.");
            return *this;
        }

        iterator operator--(int) {
            iterator old = *this;
            --(*this);
            return old;
        }

        iterator &operator+(int addition){

            for(int i = 0; i<addition;i++){
                ++(*this);
```

```
            }
            return  *this;
        }

        //Operadores para realizar comparaciones
        bool operator==(const iterator &rhs) const {
            return current == rhs.current;
        }

        bool operator!=(const iterator &rhs) const {
            return !(*this == rhs);
        }

    protected:
        //apunta al nodo al que estoy trabajando en ese momento
        Node *current;
        iterator(Node *p) : current{p} {}
        //la clase amigo de acceso a los atributos privados
        friend class DLList<Object>;

        Object & retrieve() const{
            return current->data;
        }
    };

public:
    //Define la dimension de la lista
    //Cuando se llame al constructor, ya tiene que estar una cabeza y una cola
    DLList() {
        init();
    }

    //Destructor de la lista
    //Primero borra el contenido y despues la cola y la cabeza
    ~DLList() {
        clear();
        delete head;
        delete tail;
    }

    //Sirve para meter el iterador al principio o al final
    iterator begin() { return {head->next}; }
    iterator end() { return {tail}; }

    //el tamaño de la lista, para que un iterador sepa cuanto debe recorrer
    int size() const { return theSize; }
    bool empty() const { return size() == 0; }

    //Mientras no este vacia borra el objeto de en frente
    void clear() { while (!empty()) pop_front(); }

    //Si la lista esta vacia da un error, si no retorna el inicio
    Object &front() {
        if(empty())
```

```
            throw std::logic_error("List is empty.");
        return *begin();
    }

    //funcion de push por copia
    void push_front(const Object &x) {
        insert(begin(), x);
    }
    //funcion de push por referencia
    void push_front(Object &&x) {
        insert(begin(), std::move(x));
    }

    void push_back(const Object &x) {
        insert(end(), x);
    }
    //funcion de push por referencia
    void push_back(Object &&x) {
        insert(end(), std::move(x));
    }

    //elimina el valo de en frente
    void pop_front() {
        if(empty())
            throw std::logic_error("List is empty.");
        erase(begin());
    }

    //Recibe un iterador, lee esa posicion e inserta un código en la posicion que
le demos
    //este funciona por copia
    iterator insert(iterator itr, const Object &x) {
        Node *p = itr.current;
        theSize++;
        return {p->prev = p->prev->next = new Node{x, p, p->prev}};
    }

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

    void insert(int pos, const Object &x) {
        insert(get_iterator(pos), x);
    }

    iterator get_iterator(int a)
    {
        iterator it = begin();
        for(int i = 0; i != a; ++i) {
            ++it;
        }
```

```
        return it;
    }

    //recibe un iterador y borra el dato en la posicion que le digamos
    iterator erase(iterator itr) {

        if (itr == end())
            throw std::logic_error("Cannot erase at end iterator");
        Node *p = itr.current;
        iterator returnValue(p->next);
        p->prev->next = p->next;
        p->next->prev = p->prev;
        delete p;
        theSize--;

        return returnValue;
    }

    void erase(int pos)
    {
        erase(get_iterator(pos));
    }

    //Getter para toda la lista
    void print() {
        iterator itr = begin();
        while (itr != end()) {
            std::cout << *itr << " ";
            ++itr;
        }
        std::cout << std::endl;
    }


protected:
    Node *head;
    Node *tail;
    int theSize;
    //init necesita acceso a los datos privados para inicializar una lista vacia
    void init() {
        head = new Node;
        tail = new Node;
        theSize = 0;

        head->next = tail;
        head->prev = nullptr;

        tail->prev = head;
        tail->next = nullptr;
    }


};
```

```
#endif //DLLIST_DLLIST_H
```

Stack

```
#include <iostream>
#include <cstdlib>
#include <stack>
#ifndef PROYECTO_SEGUNDO_PARCIAL_STACK_H
#define PROYECTO_SEGUNDO_PARCIAL_STACK_H

template <typename Iterator>
        class Stack : private DLList<Iterator> { // se crea una clase para stack y
hereda de DLList
        public:
            Stack() { // se crea el constructor simple
            }

            ~Stack(){ // se crea el destructor
                clear();
                delete DLList<Iterator>::head;
                delete DLList<Iterator>::tail;
            }

            void push(Iterator &data){ // inserta el dato al inicio del stack
                DLList<Iterator>::push_front(data);
            }
            void push(Iterator &&data){  // inserta por referencia el dato al
inicio del Stack
                DLList<Iterator>::push_front(data);
            }

            void pop(){  // borra el dato al inicio de la lista
                DLList<Iterator>::pop_front();
            }

            void clear(){ // borra toda la lista
                DLList<Iterator>::clear();
            }

            void print(){ // se imprime todo
                DLList<Iterator>::print();
            }

            Iterator top(){ // imprime solamente el primero
                return DLList<Iterator>::head->next->data;
            }
        };

#endif //PROYECTO_SEGUNDO_PARCIAL_STACK_H
```

Queu

```
#ifndef PROYECTO_SEGUNDO_PARCIAL_QUEUE_H
#define PROYECTO_SEGUNDO_PARCIAL_QUEUE_H
// enqueue copia referencia r y l
// dequeue
//front
// print

template<typename ZL>
class Queue : private DLList<ZL>{ // se crea una clase para queu y hereda de
DLList
public:
    Queue(){ // se crea el constructor simple
    }

    ~Queue(){ // se crea el destructor
        clear();
        delete DLList<ZL>::head; // se borra el head heredado del DLList
        delete DLList<ZL>::tail; // se borra el tail heredado del DLList
    }

    void enqueue(ZL &data){ // inserta el dato al final del queu
        DLList<ZL>::push_back(data);
    }

    void enqueue(ZL &&data){ // inserta por referencia el dato al final del queu
        DLList<ZL>::push_back(data);
    }

    void dequeue(){  // borra el dato al inicio de la lista
        DLList<ZL>::pop_front();
    }

    void print(){ // se imprime todo
        DLList<ZL>::print();
    }

    void clear(){ // borra toda la lista
        DLList<ZL>::clear();
    }

    ZL front(){ // imprime solamente el primero
        return DLList<ZL>::head->next->data;
    }
};

#endif //PROYECTO_SEGUNDO_PARCIAL_QUEUE_H
```