Proyecto progra

general view

Aqui se vera el proyecto completo con de el alumno Angel Jared Osorno Lares. El cual se basa en implementar los temas vistos durante el segundo parcial.

main

```
#include <iostream>
#include "DLList.h"
#include "Stack.h"
#include "Queue.h"
#include "SLList.h"

int main() {
    // Test Simple List
    SLList<int> simpleList;
    simpleList.push_front(20);
    simpleList.push_front(10);
    simpleList.push_front(30);
    simpleList.push_front(40);
    simpleList.push_front(50);
    std::cout << "Simple list: ";
    simpleList.print();
    simpleList.pop_front();
    std::cout << "Simple list after pop: ";
    simpleList.print();
    simpleList.clear();
    std::cout << "Simple list after clear: ";
    simpleList.print();


    // Test integer Double Linked List
    DLList<int> integerList;
    integerList.push_back(10);
    integerList.push_front(20);
    integerList.insert(1, 40);
    int a =3;
    integerList.insert(integerList.begin()+2, 30);
    integerList.insert(a, 12);
    std::cout << "Integer list: ";
    integerList.print();
    (void)integerList.erase(integerList.begin() + 1);
    std::cout << "Integer list after erase: ";
    integerList.print();
    integerList.clear();
    std::cout << "Integer list after clear: ";
    integerList.print();

    // Test double Double-Linked List
    DLList<double> doubleList;
    doubleList.push_back(1.5);
    doubleList.push_front(2.5);
    doubleList.insert(1, 3.5);
    doubleList.insert(doubleList.begin()+2, 4.5);
    doubleList.insert(2, 5.5);
    std::cout << "\nDouble list: ";
    doubleList.print();
    doubleList.erase(doubleList.begin() + 1);
    std::cout << "Double list after erase: ";
    doubleList.print();
    doubleList.clear();
    std::cout << "Double list after clear: ";
    doubleList.print();

    // Test char Double Linked List
    DLList<char> charList;
    charList.push_back('a');
    charList.push_front('b');
    charList.insert(1, 'c');
    std::cout << "\nChar list: ";
    charList.print();
    charList.erase(charList.begin() + 1);
    std::cout << "Char list after erase: ";
    charList.print();
    charList.clear();
    std::cout << "Char list after clear: ";
    charList.print();

    //Boolean list
    DLList<bool> boolList;
    boolList.push_back(true);
    boolList.push_front(false);
    boolList.insert(1, true);
    std::cout << "\nBoolean list: ";
    boolList.print();
    boolList.erase(boolList.begin() + 1);
    std::cout << "Boolean list after erase: ";
    boolList.print();
    boolList.clear();
    std::cout << "Boolean list after clear: ";
    boolList.print();

    //Test int Stack
    Stack<int> stack;
    stack.push(1);
    stack.push(2);
    stack.push(3);
    std::cout << "\nStack: ";
    stack.print();
    stack.pop();
    std::cout << "Stack after pop: ";
    std::cout << stack.top() << std::endl;

    //Test double Stack
    Stack<double> doubleStack;
    doubleStack.push(1.5);
    doubleStack.push(2.5);
    doubleStack.push(3.5);
    std::cout << "\nDouble Stack: ";
    doubleStack.print();
    doubleStack.pop();
    std::cout << "Double Stack after pop: ";
    std::cout << doubleStack.top() << std::endl;
    doubleStack.clear();

    //Test Queue
    Queue<int> queue;
    queue.enqueue(1);
    queue.enqueue(2);
    queue.enqueue(3);
    std::cout << "\nQueue: ";
    queue.print();
    queue.dequeue();
    std::cout << "Queue after dequeue: ";
    std::cout << queue.front() << std::endl;
    queue.clear();

    /*
     * This is completely optional, but if you want a challenge, try to implement
  an explicit constructor for the Data Structure classes that takes in an initialiser
  list. For example, you should be able to do something like this:     * perdon profe
  me supero     * DLList<int> list = {1, 2, 3, 4, 5};     * Stack<int> stack = {1, 2,
  3, 4, 5};     * Queue<int> queue = {1, 2, 3, 4, 5};     * SLList<int> simpleList =
  {1, 2, 3, 4, 5};
     * For anyone able to do this, you will get 20 bonus points on the assignment.
  You can do this for any of the Data Structures, but you only get the bonus points
  once.     */
    return 0;
  }
```

SLList

```
#ifndef SLLLIST_H
#define SLLLIST_H
#include <iostream>
#include <utility>

template <typename Object>
class SLList {
private:
    //Cada Cuadrito
    struct Node  {
        Object data;
        Node *next;//Anya

        Node(const Object &d = Object{}, Node *n = nullptr)
                : data{d}, next{n} {}

        Node(Object &&d, Node *n = nullptr)
                : data{std::move(d)}, next{n} {}
    };

public:
    class iterator {
    public:
        iterator() : current{nullptr} {}

        Object &operator*() {
            if(current == nullptr)
                throw std::logic_error("Trying to dereference a null pointer.");
            return current->data;
        }

        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;
        }

        bool operator==(const iterator &rhs) const {
            return current == rhs.current;
        }

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

    private:
        Node *current;
        iterator(Node *p) : current{p} {}
        friend class SLList<Object>;
    };

public:
    SLList() : head(new Node()), tail(new Node()), theSize(0) {
        head->next = tail;
    }

    ~SLList() {
        clear();
        delete head;
        delete tail;
    }

    iterator begin() { return {head->next}; }
    iterator end() { return {tail}; }

    int size() const { return theSize; }
    bool empty() const { return size() == 0; }

    void clear() { while (!empty()) pop_front(); }

    Object &front() {
        if(empty())
            throw std::logic_error("List is empty.");
        return *begin();
    }

    void push_front(const Object &x) { insert(begin(), x); }
    void push_front(Object &&x) { insert(begin(), std::move(x)); }

    void pop_front() {
        if(empty())
            throw std::logic_error("List is empty.");
        erase(begin());
    }

    iterator insert(iterator itr, const Object &x) {
        Node *p = itr.current;
        head->next = new Node{x, head->next};
        theSize++;
        return iterator(head->next);
    }

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

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

```
private:
    Node *head;
    Node *tail;
    int theSize;
    void init() {
        theSize = 0;
        head->next = tail;
    }
};

#endif //PROYECTO_SEGUNDO_PARCIAL_SLLIST_H
```

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 ZL>
        class Stack : private DLList<ZL> {
        public:
            // constructores
            Stack()
            {
            DLList<ZL>::init();
            }
            void push(ZL &data)
            {
                DLList<ZL>::push_front(data);
            }
            void push(ZL &&data)
            {
                DLList<ZL>::push_front(data);
            }
            // destructor
            ~Stack()
            {
                clear();
                delete DLList<ZL>::head;
                delete DLList<ZL>::tail;
            }
            // Eliminar el dato de hasta adelante
            void pop()
            {
                DLList<ZL>::pop_front();
            }
            // Limpia la stack
            void clear()
            {
                DLList<ZL>::clear();
            }
            // imprime el stack
            void print()
            {
                DLList<ZL>::print();
            }
            // muestra el primer dato
            ZL top()
            {
                return DLList<ZL>::head->next->data;
            }
        };

#endif //PROYECTO_SEGUNDO_PARCIAL_STACK_H
```

Queque

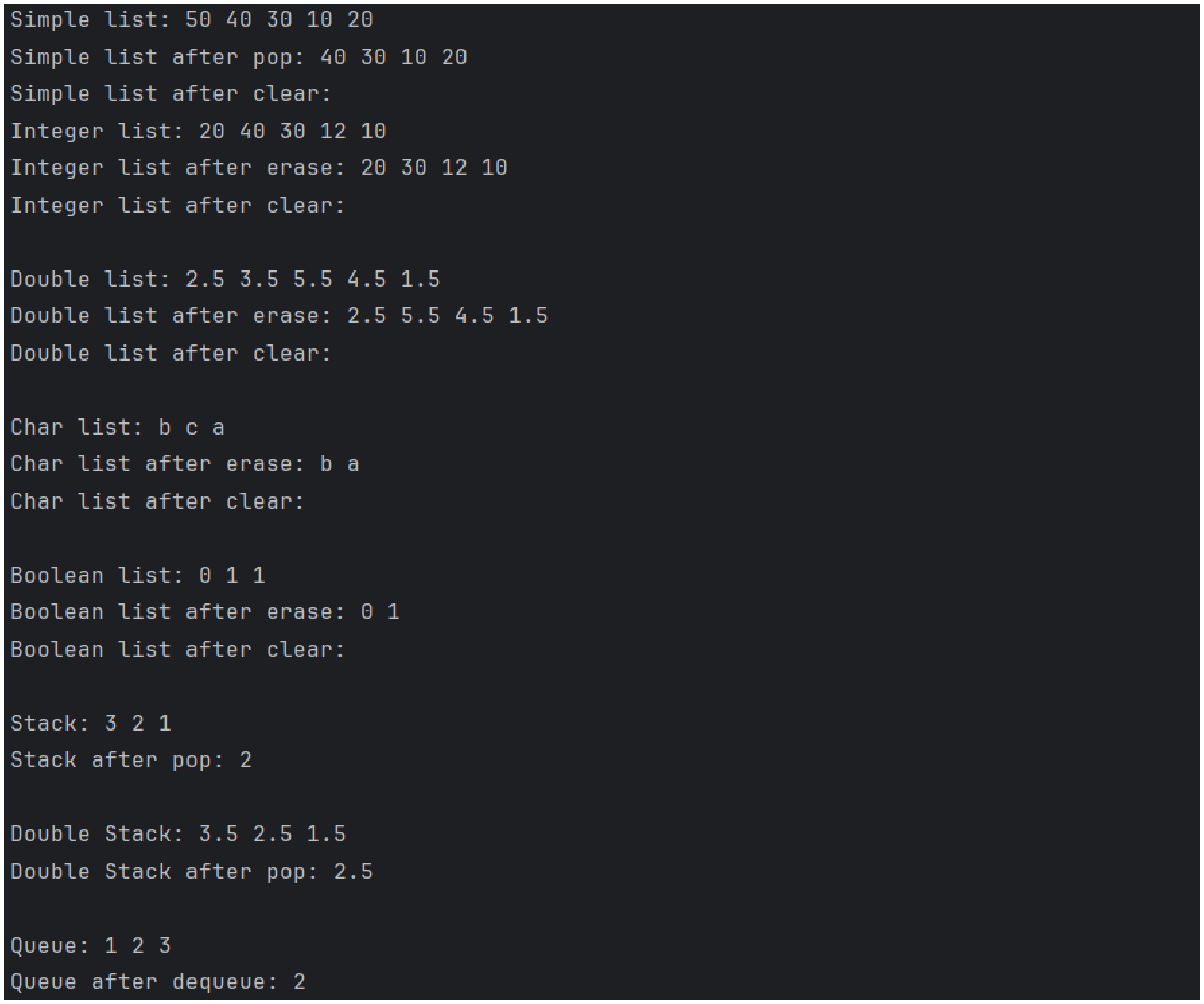
```
#ifndef PROYECTO_SEGUNDO_PARCIAL_QUEUE_H
#define PROYECTO_SEGUNDO_PARCIAL_QUEUE_H

template<typename ZL>
class Queue : private DLList<ZL> {
public:
    // constructores
    Queue()
        {
        DLList<ZL>::init();
        }
    void enqueue(ZL &data) {
        DLList<ZL>::push_back(data);
    }
    void enqueue(ZL &&data) {
        DLList<ZL>::push_back(data);
    }
    // destructor
    ~Queue() {
        clear();
        delete DLList<ZL>::head;
        delete DLList<ZL>::tail;
    }
    // Elimina el primer dato en la cola
    void dequeue() {
        DLList<ZL>::pop_front();
    }

    // Imprime la cola
    void print()
    {
        DLList<ZL>::print();
    }
    // elimina la cola
    void clear()
    {
        DLList<ZL>::clear();
    }
    // muestra el primer elemto de la cola
    ZL front()
    {
        return DLList<ZL>::head->next->data;
    }
};

#endif //PROYECTO_SEGUNDO_PARCIAL_QUEUE_H
```

ejecución



```
Simple list: 50 40 30 10 20
Simple list after pop: 40 30 10 20
Simple list after clear:
Integer list: 20 40 30 12 10
Integer list after erase: 20 30 12 10
Integer list after clear:

Double list: 2.5 3.5 5.5 4.5 1.5
Double list after erase: 2.5 5.5 4.5 1.5
Double list after clear:

Char list: b c a
Char list after erase: b a
Char list after clear:

Boolean list: 0 1 1
Boolean list after erase: 0 1
Boolean list after clear:

Stack: 3 2 1
Stack after pop: 2

Double Stack: 3.5 2.5 1.5
Double Stack after pop: 2.5

Queue: 1 2 3
Queue after dequeue: 2
```