vector dinamico con template:

```
template <class T>
class Vector{
	public:
		Vector();
		void push_back(T data); //agregar elemento
		T get(int pos); //recuperar elemento
		void eliminar2(T dato); //eliminar
		int size() { return this->tamanio; } //nose pero metodo util
	private:
		void resize(int newMax);
		T * data;
		int max;
		int tamanio;
};
template<class T>
Vector<T>::Vector() {
	this->data = new T[2];
	this->max = 2;
	this->tamanio = 0;
}
template<class T>
void Vector<T>::push_back(T data2) {
	if (tamanio == max){
		this->resize(this->max * 2);
	}
	this->data[tamanio++] = data2;
}
template<class T>
T Vector<T>::get(int pos) {
	return data[pos];
}
template<class T>
T& Vector<T>::operator[](int pos) {
	return data[pos];
}
template<class T>
void Vector<T>::resize(int newMax) {
	T * temp = new T[newMax];
	for (int i = 0; i < this->max; i++){
		temp[i] = this->data[i];
	}
	delete[] this->data;
	this->data = temp;
	this->max = newMax;
}
template <class T>
void Vector<T>::eliminar(T dato){
	int j = 0;
	for (int i = 0; i < this-> tamanio; i++) {
		if (this->data[i] != dato) {
			this->data[j++] = this->data[i];
```

```
                }
}
        this-> tamanio = j;
}
```

lista doblemente enlazada con template:

```
template <class T>
class ListaDE {
        public:
                ListaDE();
                ListaDE(T dato);
                ~ListaDE();
                void insertar(T dato);
                bool eliminar(T dato);
                void obtener(T dato);
                int getSize() const;
                T operator[] (int index)
        private:
                struct nodo{
                        T dato;
                        int index; //no es necesario
                        nodo * siguiente = nullptr;
                        nodo * anterior = nullptr;
                };
                nodo * frente;
                nodo * fondo;
                int size;
                void deleteNodo(nodo * nodo2eliminar);
};
template<class T>
ListaDE<T>::ListaDE() {
        frente = fondo = nullptr;
        this->size = 0;
}
template<class T>
ListaDE<T>::ListaDE(T dato) {
        this->size = 0;
        insertar(dato);
}
template<class T>
ListaDE<T>::~ListaDE() {
        if (frente) {
                nodo * aux = frente;
                nodo * aux2 = aux;
                while (aux) {
                        aux2 = aux->siguiente;
                        delete aux;
                        aux = aux2;
                }
        }
}
template<class T>
void ListaDE<T>::insertar(T dato) {
```

```
	if (size == 0){
		frente = fondo = new nodo{dato, 0};
		this->size = 1;
	} else {
		this->fondo->siguiente = new nodo{dato, this->size};
		this->fondo = this->fondo->siguiente;
		size++;
	} //ciera el else } //cierra la funcion
template<class T>
bool ListaDE<T>::eliminar(T _dato) {
	nodo * aux = frente;
	while (aux != nullptr) {
		if (aux->dato == _dato) {
			deleteNodo(aux);
			return true;
		}
	}
	return false;
}
template<class T>
void ListaDE<T>::deleteNodo(nodo *nodo2eliminar) {
	nodo * aux = nodo2eliminar->siguiente;
	while (aux) {
		aux->index--;
		aux = aux->siguiente;
	}
	if (aux->anterior) {
		aux->anterior->siguiente = aux->siguiente;
	}
	if (aux->siguiente) {
		aux->siguiente->anterior = aux->anterior;
	}
	delete aux;
	this->size--;
}
template <class T>
int ListaDE<T>::getSize() const {
	return this->size;
}
template<class T>
T ListaDE<T>::operator[] (int index) {
	nodo * aux = frente;
	while (aux != nullptr) {
		if (aux->index == index) {
			return aux->dato;
		}
	}
	return T{};
}
```

pila con template:

```
template <class T>
```

```
class pila {
	public:
		pila();
		pila(T dato);
		~pila();
		void push(T newDato);
		T pop();
		int getSize() const;
	private:
		struct nodo {
		T dato;
		nodo * link;
	};
	nodo * frente;
	int size;
};
template<class T>
pila<T>::pila() {
	frente = nullptr;
	this->size = 0;
}
template<class T>
pila<T>::pila(T dato) {
	this->frente = new nodo{dato, nullptr};
	this->size = 1;
}
template<class T>
pila<T>::~pila() {
	while(this->size)
		this->pop();
}
template<class T>
void pila<T>::push(T newDato) {
	nodo * aux = new nodo{newDato, frente};
	frente = aux;
	size++;
}
template<class T>
T pila<T>::pop() {
	if (size == 0)
		return T{}; //pila ya vacia
	nodo * aux = frente;
	frente = frente->link;
	T dato2retornar = aux->dato;
	delete aux;
	size--;
	return dato2retornar;
}
template<class T>
int pila<T>::getSize() const {
	return size;
}
```

lista con template:

```
template <class T>
class ListaSimple {
	public:
		ListaSimple();
		~ListaSimple();
		void insertarAlInicio(T dato);
		void insertarAlFinal(T dato);
		T eliminarAlInicio();
		bool estaVacia() const;
		int getSize() const;
		void imprimir() const;
	private:
		struct Nodo {
			T dato;
			Nodo* siguiente;
		};
		Nodo* cabeza;
		int size;
};
template <class T>
ListaSimple<T>::ListaSimple() {
	cabeza= NULL;
	this->size=0;
}
template <class T>
ListaSimple<T>::~ListaSimple() {
	while (!estaVacia()) {
	eliminarAlInicio();
	}
}
template <class T>
void ListaSimple<T>::insertarAlInicio(T dato) {
	Nodo* nuevoNodo = new Nodo{dato, cabeza};
	cabeza = nuevoNodo; size++;
}
template <class T>
void ListaSimple<T>::insertarAlFinal(T dato) {
	Nodo* nuevoNodo = new Nodo{dato, nullptr};
	if (estaVacia()) {
		cabeza = nuevoNodo;
	} else {
		Nodo* actual = cabeza;
		while (actual->siguiente != nullptr) {
			actual = actual->siguiente;
		} actual->siguiente = nuevoNodo;
	}
	size++;
}
template <class T>
T ListaSimple<T>::eliminarAlInicio() {
```

```
        if (estaVacia()) {
                return T{};
        }
        Nodo* temp = cabeza;
        T dato = cabeza->dato;
        cabeza = cabeza->siguiente;
        delete temp;
        size--;
        return dato;
}
template <class T>
bool ListaSimple<T>::estaVacia() const {
        return cabeza == nullptr;
}
template <class T>
int ListaSimple<T>::getSize() const {
        return size;
}
template <class T>
void ListaSimple<T>::imprimir() const {
        Nodo* actual = cabeza;
        while (actual != nullptr) {
                std::cout << actual->dato << " - ";
                actual = actual->siguiente;
        }
}
```

cola con templates:

```
template <class T>
class Cola {
        public:
                Cola();
                ~Cola();
                void add(T dato); // Agrega un elemento al final de la cola
                T eliminar(); // Elimina y devuelve el elemento
                bool estaVacia() const;
                int getSize() const;
                void imprimir() const;
        private:
                struct Nodo {
                        T dato;
                        Nodo* siguiente;
                };
        Nodo* frente;
        Nodo* final;
        int size;
};
template <class T>
Cola<T>::Cola() {
        this->frente=nullptr;
        this->final = nullptr;
        this->size = 0;
```

```
}
template <class T>
Cola<T>::~Cola() {
	while (!estaVacia()) {
		eliminar();
	}
}
template <class T>
void Cola<T>::add(T dato) {
	Nodo* nuevoNodo = new Nodo{dato, nullptr};
	if (estaVacia()) {
		frente = final = nuevoNodo;
	} else {
		final->siguiente = nuevoNodo;
		final = nuevoNodo;
	}
	size++;
}
template <class T>
T Cola<T>::eliminar() {  // //si no pide devolver poner en void
	if (estaVacia()) {
		return T{};
	}
	Nodo* temp = frente;
	T dato = frente->dato; //si no pide devolver no hace falta esta linea
	frente = frente->siguiente;
	if (frente == nullptr) { // Si la cola queda vacía actualiza final
		final = nullptr;
	}
	delete temp;
	size--;
	return dato; //si no pide devolver no hace falta esta linea
}
template <class T>
bool Cola<T>::estaVacia() const {
	return frente == nullptr;
}
template <class T>
int Cola<T>::getSize() const {
	return size;
}
template <class T>
void Cola<T>::imprimir() const {
	Nodo* actual = frente;
	while (actual != nullptr) {
		std::cout << actual->dato << " <- ";
		actual = actual->siguiente;
	}
}
```

sistema uader:

```
void redimensionar(Examen* &arrayviejo, int &max, int maxnuevo) {
```

```
        Examen* arraynuevo = new Examen[maxnuevo];
        for(int i=0;i<max;i++){
                arraynuevo[i] = arrayviejo[i];
        }
        if(arrayviejo){
                delete[] arrayviejo;
        }
        arrayviejo=arraynuevo;
        max=maxnuevo;
}
class Curso {
        public:
                Curso(int cantexamenes, int cantrecuperatorios) {};
                void agregarAlumno(Alumno, int posicion);
                Alumno getAlumno(int posicion);
                void estadoAlumno();
                //ACTIVIDAD 3
                friend std::ostream& operator << (std::ostream& os, const Curso& c) {
                        c.estadoAlumno();
                        for(int i = 0; i < c.cantalumnos; i++)
                                os << "Nombre: " << c.alumnos[i].nombre << " Estado: " <<
                        c.alumnos[i].estado << std::endl;
                        return os;
                }
        protected:
                int cantex, cantrec, cantalumnos=0;
                Alumno alumno[50];
};
void Curso::agregarAlumno(Alumno aa, int posicion) {
        alumno[posicion] = aa;
        cantalumnos++;
}
Alumno Curso::getAlumno(int posicion) {
        return alumno[posicion];
}
//ACTIVIDAD 2
void * Curso::estadoAlumno() {
        for (int i = 0; i < cantalumnos; i++) {
                int aux = alumno[i].getPromedio();
                int aux2 = alumno[i].getCantExamenes;
                if (aux2 < cantex) {
                        alumno[i].setEstado = "Libre ausente";
                } else {
                bool examenMenor50 = false;
                        for(int j=0;j<aux2;j++) {
                                if (alumno[i].examenes[j].getNota() < 50) {
                                        examenMenor50 = true;
                                        break;
                                }
                        }
                        if (aux < 55 or examenMenor50) {
                                alumno[i].setEstado("Libre");
```

```
                        } else {
                                if (aux >= 55 and aux < 75) {
                                        alumno[i].setEstado("Regular");
                                } else {
                                bool promocionado = true;
                                        for(int j=0;j<aux2;j++) {
                                                if (alumno[i].examenes[j].getNota() < 70) {
                                                        promocionado = false;
                                                        break;
                                                }
                                        }
                                        if (promocionado) {
                                                alumno[i].setEstado("Promocionado");
                                        } else alumno[i].setEstado("Regular");
                                }
                        }
                }
        }
}
class Alumno {
        public:
                Alumno(int dni, char* nya) {
                        this->nya = new char[strlen(nya)+1];
                        strcpy(this->nya, nya);
};
        void agregarExamen(Examen);
        void removerExamen(int nroex, int notita);
        char * getNombre(){
                return nya;
        }
        char * setEstado() {} //nose como se hace
        char * getEstado(){
                return estado;
        }
        int getCantExamenes(){
                return cantexameneshechos;
        }
        int promedio();
        int getPromedio(){
                return prom;
        }
  protected:
        int dni, prom=0;
        char * nya, estado;
        Examen * examenes = NULL;
        int cantexameneshechos = 0;
};
void Alumno::agregarExamen(Examen ee) {
        int indice = ee.getNroExamen()-1;
        if(ee.getTipo()=='P') {
                if(indice >= cantexameneshechos) {
                        redimensionar(examenes, cantexameneshechos, cantexameneshechos+1);
```

```
                }
                examenes[indice]=ee;
                cantexameneshechos++;
        } else examenes[indice]=ee;
}
void Alumno::removerExamen(int nroex, int notita) {
        int j = 0;
        for (int i=0;i<cantexameneshechos;i++){
                if(examenes[i].getNota() != notita && examenes[i].getNroExamen() != nroex) {
                        examenes[j] = examenes[i];
                        j++;
                }
        }
        cantexameneshechos=j;
}
int Alumno::promedio(){
        for ( int i = 0; i < cantexameneshechos; i++ ) {
                prom+=examenes[i].nota;
        }
        prom=prom/cantexameneshechos;
        return prom;
}
class Examen {
        public:
        Examen(int notita, int nroex, char tipito) {};
        int getNota() {
                return nota;
        }
        int getNroExamen() {
                return nroexamen;
        }
        char getTipo() {
                return tipo;
        }
        protected:
                int nota, nroexamen;
                char tipo;
};
```