```
    void print(ostream& fout) const {
        fout << "{ ";
        for(int i=0; i<top; i++){
            fout << container[i];
            if(i != top-1)
                fout << ", ";
        }
        fout << " }" << endl;
    }
};
template <class T>
ostream& operator<<(ostream& fout, const Set<T>& set){
    set.print(fout);
    return fout;
}
```

ES 12/01/23

```
template <class T>
class Queue {
private:
    struct node{
        T element;
        node* next;
    };
    node *first;
    node *last;
}

public:
    Queue(){
        this->first = nullptr;
        this->last = nullptr;
    }
    ~Queue(){
        while(!isEmpty())
            dequeue();
    }
    Queue (const Queue& other){
        this->first = nullptr;
        this->last = nullptr;
        node* cursor = other.first;
        while (cursor != nullptr){
                enqueue(cursor->element);
                cursor = cursor->next;
        }
    }
    Queue& operator=(const Queue& other){
        if(this != &other){
            while(!isEmpty())
                dequeue();
            this->first = nullptr;
            this->last = nullptr;
            node* cursor = other.first;
```

```
        while (cursor != nullptr){
            enqueue (cursor -> element);
            cursor = cursor ->next;
        }
    return *this;
}

void enqueue (const T& elem){
    node* newNode = new node;
    newNode -> element= elem;
    newNode ->next= nullptr;
    if (isEmpty()){
        last = newNode;
        firs= newNode;
    }
    last-> next = newNode;
    last = newNode;
}

T dequeue (){
    if (isEmpty())
       throw runtime_error("cola vuota");
    node* tmp = first;
    T result = first -> element;
    first = first -> next;
    if (first == nullptr)
      last = nullptr;
    delete tmp;
    return result;
}

bool isEmpty() const {
    return first == nullptr;
}

void print (ostream& fout) const{
    fout << "{";
    node * cursor = first;
    while (cursor != nullptr){
        fout << cursor -> element;
```

```
            if (cursor->next != nullptr)
                fout << ", ";
            cursor = cursor->next;
        }
        fout << "}" << endl;
    }
};
template <class T>
ostream& operator<<(ostream& fout, const Queue<T>& queue) {
    queue.print(fout);
    return fout;
}
```

Es. 2 12/01/23

```
public class Stack<T> implements ComparableStack<T>> {
    private ArrayList<T> container;

    public Stack(){
        this.container = new ArrayList<>();
    }
    public void push(T elem){
        container.add(elem);
    }
    public T pop() throws EmptyStackException {
        if (isEmpty())
            throw new EmptyStackException("stack vuoto");
        return container.remove(container.size()-1);
    }
    public T peek() throws EmptyStackException{
        if (isEmpty())
            throw new EmptyStackException("Stack vuoto");
        return container.get(container.size()-1) ;
    }
    public boolean isEmpty(){
        return container.isEmpty();
    }
    public int size(){
        return container.size();
    }
    public String toString(){
        return container.toString();
    }
    public boolean equals(Object obj){
        if(this == obj)
            return true;
        else if (obj == null)
            return false;
        else if(getClass() != obj.getClass())
            return false;
```

```
        Stack<?> other = (Stack<?>) obj;
        return container.equals(other.container);
    }
    public int compareTo(Stack<?> other) {
        return Integer.compare(this.size(), other.size());
    }
}

public class EmptyStackException extends Exception {
    public EmptyStackException(String msg) {
        super(msg);
    }
}
```

Es. 1 10-06-22

```
template <class T>
class Insieme<T>{
private:
    T* container;
    int dim;
    int top;
    void resize(){
        T* tmp= new T[dim*2];
        for(int i=0; i<dim; i++)
            tmp[i] = container[i];
        delete[] this->container;
        this->dim *=2;
        this->container = tmp;
    }
public:
    Insieme(){
        this->dim = 10;
        this->container = new T[dim];
        this->top=0;
    }
    ~Insieme(){
        delete[] this->container;
    }
    Insieme(const Insieme& other){
        this->dim = other.dim;
        this->container = new T[dim];
        this->top = other.top;
        for(int i=0; i< top ; i++)
            container[i] = other.container[i];
    }
    Insieme& operator=(const Insieme& other){
        if(this != &other){
            delete[] this.container;
            this->dim = other.dim;
            this->top = other.top;
```

```
        this.container = new T[dim];
        for (int i=0; i<top; i++)
            container[i] = other.container[i];
    }
    return *this;
}
void add (T elem){
    if (contains(elem))
        throw runtime_error ("elemento gia' presente");
    if (top == dim)
        resize();
    this->container[top++] = elem;
}
bool contains(T elem) const {
    for(int i=0; i<top; i++){
        if(container[i] == elem)
            return true;
    }
    return false;
}
int size() const {
    return top;
}
Insieme & operator + (const Insieme& other) {
    Insieme result;
    for(int i=0; i<top; i++)
        result.add(container[i]);
    for(int i=0; i<other.top; i++)
        if(!(result.contains(other.container[i])))
            result.add(other.container[i]);
    return result;
}
void print(ostream& out) const{
    out << "{ ";
    for(int i=0; i<top; i++){
        out << container[i];
        if(i!=top-1)
```

```
            fout << ", ";
        }
        fout << " }";
    }
};
template <class T>
ostream& operator<<(ostream& fout, const Insieme<T>& ins){
    ins.print(fout);
    return fout;
}
```

ES 2 10-06-22

```
public class GreenPass{
    protected String codiceFiscale;
    protected Data scadenza;
    public GreenPass (String codice, Data scadenza){
        this.codiceFiscale = codice;
        this.scadenza= scadenza;
    }
    public String toString( ){
        return "codice fiscale: " + codiceFiscale +" "
                + "data di scadenza: " + scadenza;
    }
    public boolean equals(Object obj){
        if (this == obj)
            return true;
        else if (obj== null)
            return false;
        else if (getClass() != obj.getClass())
            return false;
        GreenPass other = (GreenPass) obj;
        return codiceFiscale.equals(other.codiceFiscale)
                && scadenza.equals(other.scadenza);
    }
}
public class GreenPassVaccino extends GreenPass{
    public enum Vaccino{
        VACCINO1,
        VACCINO2,
        VACCINO3
    }
    private Vaccino tipoVaccino;
    public GreenPassVaccino(String codFisc, String scadenza, Vaccino tipovacc){
        super(codFis, scadenza);
        this.tipoVaccino = tipovacc;
    }
```

```
    public String toString(){
        return super.toString + tipoVaccino;
    }

    public boolean equals(Object obj){
        if (!super.equals(obj))
            return false;
        GreenPassVaccino other = (GreenPassVaccino) obj;
        return tipoVaccino.equals(other.tipoVaccino);
    }
                tipoVaccino == other.tipoVaccino
}

public class GreenPassTampone extends GreenPass{
    public enum Tampone{
        TAMPONE1,
        TAMPONE2,
        TAMPONE3
    }

    public Tampone tipoTampone;
    public GreenPassTampone(String codFisc, String scadenza, Tampone tipoTamp){
        super(codFisc, scadenza);
        this.tipoTampone = tipoTamp;
    }

    public String toString(){
        return super.toString+" "+tipoTampone;
    }

    public boolean equals(Object obj){
        if(!super.equals(obj))
            return false;
        GreenPassTampone other = (GreenPassTampone) obj;
        return tipoTampone.equals(other.tipoTampone);
    }
                tipoTampone == other.tipoTampone
}

public class PersonaConGreenPass{
    private Set<GreenPass> greenPass;
    private String codFiscale;
    private static boolean noGPVaccino = false;
    public PersonaConGreenPass(String codFisc){
```

```
        this.codFiscale = codFisc;
        this.greemPass = new HashSet<>();
    }

    public void addGreemPass(GreemPass gP) throws GreemPassException
        if (gP instanceof GreemPassVaccino){
            if ( mGPvaccino )
                throw new GreemPassException(" ");
            else if ( gP.codiceFiscale.equals(codFiscale))
                throw new GreemPassException(" ");
            greemPass.add(gP);
            mGPvaccino= true;
        }
        if (gP instanceof GreemPassTampone){
            if (gP.codiceFiscale.equals(codFiscale))
                throw new GreemPassException(" ");
            greemPass.add(gP);
        }
    }
    public String toString(){
        return codFiscale + " " + greemPass;
    }
    public boolean equals(Object obj){
        if (this== obj)
            return true;
        else if ( obj == null )
            return false;
        else if (getClass() != obj.getClass())
            return false;
        PersonaConGreemPass other = (PersonaConGreemPass) obj;
        return codFiscale.equals(other.codFiscale)
                && greemPass.equals(other.greemPass);
    }
}
```

ES. 1 01/09/23

```
template <class T>
class Set{
private:
    T* container;
    int dim;
    int size;
    void resize(){
        T* tmp = new T[dim*2];
        for(int i=0; i<dim; i++)
            tmp[i]= container[i];
        this->dim *= 2;
        delete [] this->container;
        this->container = tmp;
    }
public:
    Set(){
        this->dim= 10;
        this->container = new T[dim];
        this->size= 0;
    }
    ~Set(){
        delete [] this-> container;
    }
    Set(const Set& other){
        this->dim = other.dim;
        this->size= other.size;
        this->container= new T[dim];
        for(int i=0; i<top; i++)
            this->container[i] = other.container[i];
    }
    Set& operator=(const Set& other){
        if (this != &other){
            delete [] this->container;
            this->dim= other.dim;
            this->size = other.size;
```

```
        this->container = new T[dim];
        for(int i=0; i<size; i++)
            this->container[i] = other.container[i];
    }
    return *this;
}

void add(T elem){
    if (contains(elem))
        throw runtime_error(" ");
    if (dim == size)
        resize();
    this->container[size++] = elem;
}

bool contains(T elem) const {
    for (int i=0; i<size; i++){
        if (container[i] == elem)
            return true;
    return false;
}

int size() const {
    return size;
}

Set<T>&                    Set<T>&
Set& operator - (const Set other){
    Set result;
    for (int i=0; i<size ; i++){
        if (contains(other.container[i]))
            result.add(container[i]);
    }
    return result;
}

void print(ostream& fout) const {
    fout << "{ ";
    for (int i=0; i<size; i++){
        fout << container[i];
        if(i != size-1)
            fout << ", ";
```

```
        }
        fout << " }";
    }
};
template <class T>
ostream& operator<< (ostream& fout, const Set<T>& set){
    set.print(fout);
    return fout;
}
```

ES. 2 02/09/23

```
public abstract class Eleggibile {
    private String nome;
    protected int nVoti;
    public Eleggibile(String nome, int nVoti){
        this.nome = nome;
        this.nVoti = nVoti;
    }
    public String getNome(){
        return nome;
    }
    public int getVoti(){
        return nVoti;
    }
    public boolean equals(Object obj){
        if(this == obj)
            return true;
        else if (obj == null)
            return false;
        else if (getClass() != obj.getClass())
            return false;
        Eleggibile other = (Eleggibile) obj;
        return nVoti == other.nVoti;
                && nome.equals(other.nome);
    }
}

public class Partito extends Eleggibile{
    public Partito(String nome){
        super(nome, 0);
    }
    public void vota(){
        nVoti++; super.nVoti++;
    }
    public boolean equals(Object obj){
        return super.equals(obj);
    }
```

```
}

public class Coalizione extends Eleggibile implements Iterable<Partito>{
    private Set<Partito> partiti;
    public Coalizione(String nome, Partito[] part){
        super(nome,0);
        for(Partito partito: part){
            this.nVoti += partito.getVoti();
            partiti.add(partito);
    }
    public int getVoti(){
        int votiTot=0;
        for(Partito parTito: partiti)
            votiTot += partito.getVoti();
        return votiTot;
    }
    public boolean equals(Object obj){
        if(!super.equals(obj))
            return false;
        Coalizione other=(Coalizione) obj;
        return partiti.equals(other.partiti);
    }
    public Iterator<Partito> iterator(){
        return partiti.iterator();
    }
}
public class Elezione{
    private Set<Coalizione> coalizioni;
    public Elezione(){
        this.coalizioni = new HashSet<>();
    }
    public void add(Coalizione coalizione){
        if(coalizioni.contains(coalizione)
            throw new RuntimeException(" ");
        coalizioni.add(coalizione);
    }
```

```
    public Coalizione vincitore(){
        Coalizione vincitore = null;
        for(Coalizione coalizione : coalizioni){
            if(vincitore == null || vincitore.getVoti() < coalizione.getVoti())
                vincitore = coalizione;
        }
        return vincitore;
    }
}
```

Es. 2 03-02-23

```
public interface MetodoPagamento{
    double verificaSaldo();
    void incrementa(double value);
    void decrementa(double value); → throws SaldoNonSufficienteException;
}

public class CartaDiCredito implements MetodoPagamento{
    private String numeroCarta;
    private double saldo;
    public CartaDiCredito(String numeroCarta, double saldo){
        this.numeroCarta = numeroCarta;
        this.saldo = saldo;
    }
    public double verificaSaldo(){
        return saldo;
    }
    public void incrementa(double value){
        this.saldo += value;
    }
    public void decrementa(double value) throws SaldoNonSufficienteException{
        if (value > saldo)
            throw new SaldoNonSufficienteException(" ");
        this.saldo -= value;
    }
    public boolean equals(Object obj){
        if(this == obj)
            return true;
        else if(obj == null)
            return false;
        else if (getClass() != obj.getClass())
            return false;
        CartaDiCredito other = (CartaDiCredito) obj;
        return numeroCarta.equals(other.numeroCarta)
                && saldo == other.saldo;
    }
}
```

```
public class ContoCorrente implements MetodoPagamento, implements Comparable<ContoCorrente>{
    private String IBAN;
    private double saldo;
    public ContoCorrente(String IBAN, double saldo){
        this.IBAN = IBAN;
        this.saldo = saldo;
    }
    public double verificaSaldo(){
        return saldo;
    }
    public void incrementa(double value){
        this.saldo += value;
    }
    public void decrementa(double value) throws SaldoNonSufficienteException{
        if(value > saldo)
            throw new SaldoNonSufficienteException(" ");
        this.saldo -= value;
    }
    public boolean equals(Object obj){
        if(this == obj)
            return true;
        else if(obj == null)
            return false;
        else if(getClass() != obj.getClass())
            return false;
        ContoCorrente other = (ContoCorrente) obj;
        return IBAN.equals(other.IBAN)
                && saldo == other.saldo;
    }
}
public class SaldoNonSufficienteException extends Exception{
    public SaldoNonSufficienteException(String msg){
        super(msg);
    }
}
```

```
public int compareTo(ContoCorrente other){
    return Double.compare(saldo, other.saldo);
}
```