

Lanfranco Lopriore

**ESERCIZI DI PROGRAMMAZIONE
SUI TIPI DI DATI ASTRATTI
UTILIZZANDO IL LINGUAGGIO C++**

*con contributi di Cinzia Bernardeschi
e Francesco Potortì*

Febbraio 1999

ESERCIZI SVOLTI[‡]

Esercizio n. 1

Una Ruota è divisa in spicchi, ciascuno dei quali può essere opaco o trasparente. Le operazioni che possono essere effettuate su una ruota sono le seguenti:

- Ruota `r()`
Costruttore di default, che inizializza una ruota `r` formata da un solo spicchio. Tale spicchio è trasparente.
- Ruota `r(n)`
Costruttore che inizializza una ruota `r` formata da `n` spicchi. Tali spicchi sono trasparenti.
- Ruota `r1(r)`
Costruttore di copia, che inizializza una ruota `r1` col valore della ruota `r`.
- `r1 = r`
Operatore di assegnamento, che sostituisce il valore della ruota risultato `r1` con quello della ruota `r`.
- `r.rendiOpaco(s)`
Operazione che rende opaco lo `s`-esimo spicchio della ruota `r`.
- `r.rendiTrasparente(s)`
Operazione che rende trasparente lo `s`-esimo spicchio della ruota `r`.
- `r.seiOpaco(s)`
Operazione che ritorna 1 se lo `s`-esimo spicchio della ruota `r` è opaco, e 0 altrimenti.
- `r.gira(p)`
Operazione che gira la ruota `r` in avanti di `p` posizioni (cioè, il primo spicchio diventa il `(p+1)`-esimo, il secondo spicchio diventa il `(p+2)`-esimo, e così via).
- `r1.sovrapponi(r)`
Operazione che modifica la ruota `r1` sovrapponendo a ciascuno spicchio di `r1` il corrispondente spicchio della ruota `r`. La sovrapposizione di due spicchi è trasparente se ambedue tali spicchi sono trasparenti, altrimenti la sovrapposizione è opaca. La ruota `r` deve avere lo stesso numero di spicchi della ruota `r1`.
- `r.spicchi(m)`
Operazione che modifica la ruota `r` in modo tale che essa diventi composta da `m` spicchi. Se inizialmente `r` ha *più* di `m` spicchi, l'operazione elimina gli spicchi a *bassi* numeri d'ordine. Se inizialmente `r` ha *meno* di `m` spicchi, l'operazione aggiunge nuovi spicchi ad *alti* numeri d'ordine; tali nuovi spicchi sono trasparenti. Se inizialmente `r` ha esattamente `m` spicchi, l'operazione lascia la ruota inalterata.
- `~Ruota()`
Distruttore.

[‡] La soluzione di alcuni esercizi comprende una funzione `main`, che ha l'unico scopo di permettere l'esecuzione delle principali funzionalità del tipo di dati astratti definito nell'esercizio stesso.

I programmi inclusi in questa dispensa hanno valore esclusivamente didattico. Essi sono stati verificati con cura, ma non sono garantiti per nessuno scopo specifico. Gli autori non si assumono nessuna responsabilità riguardo ad essi.

- `cout << r`

Operatore di uscita per il tipo `Ruota`. L'uscita ha la forma seguente:

```
<10> --XXX-X-X-
```

Le parentesi angolate racchiudono il numero degli spicchi che compongono la ruota `r`, il carattere `'-'` rappresenta uno spicchio trasparente, ed il carattere `'X'` rappresenta uno spicchio opaco.

Utilizzando il linguaggio C++, realizzare il tipo di dati astratti `Ruota`, definito dalle precedenti specifiche, in modo tale che sia possibile concatenare le operazioni `rendiOpaco()`, `rendiTrasparente()`, `gira()`, `sovrapponi()` e `spicchi()`; esempio: `r1.spicchi(m).sovrapponi(r)`. Fare ricorso ad una rappresentazione della ruota a lista.

Soluzione

```

/*****
 *
 *                               Ruota.h
 *
 *****/

class Ruota {
public:
    friend ostream& operator<<(ostream&, const Ruota&);
    enum statoDiSpicchio {TRASPARENTE, OPACO};
    Ruota(int = 1);
    Ruota(const Ruota&);
    Ruota& operator=(const Ruota&);
    Ruota& rendiOpaco(int);
    Ruota& rendiTrasparente(int);
    int seiOpaco(int s) const;
    Ruota& gira(int);
    Ruota& sovrapponi(const Ruota&);
    Ruota& spicchi(int m);
    ~Ruota();
private:
    class Spicchio {
        friend class Ruota;
        friend ostream& operator<<(ostream&, const Ruota&);
        statoDiSpicchio stato;
        Spicchio *prossimo;
    public:
        Spicchio(statoDiSpicchio st = TRASPARENTE, Spicchio *pr = 0)
            : stato(st), prossimo(pr) {}
    };
    int quanti;
    Spicchio* ultimo;
    Spicchio& operator[](int) const;
    void contraì(int);
    void espandi(int);
};

/*****
 *
 *                               Ruota.cp
 *
 *****/

#include <iostream.h>
#include <stdlib.h>
#include "Ruota.h"

void errore(const char* s)
{
    cerr << "Ruota: " << s << '\n';
    exit(1);
}

```

```

Ruota::Spicchio& Ruota::operator[](int s) const
{
    Spicchio *p = ultimo;
    for (int i = 0; i < s; i++)
        p = p->prossimo;
    return *p;
}

void Ruota::contrai(int n)
{
    Spicchio *p = ultimo->prossimo;
    Spicchio *q = p->prossimo;
    for (int i = 0; i < n; i++) {
        q = p->prossimo;
        delete p;
        p = q;
    }
    ultimo->prossimo = p;
    quanti -= n;
}

void Ruota::espandi(int n)
{
    Spicchio *primo = ultimo->prossimo;
    Spicchio *p = ultimo;
    for (int i = 0; i < n; i++) {
        Spicchio *q = new Spicchio;
        p->prossimo = q;
        p = q;
    }
    ultimo = p;
    ultimo->prossimo = primo;
    quanti += n;
}

Ruota::Ruota(int n)
{
    if (n <= 0)
        errore("Dimensione di ruota negativa o nulla");
    ultimo = new Spicchio;
    Spicchio* p = ultimo;
    for (int i = 1; i < n; i++) {
        p->prossimo = new Spicchio;
        p = p->prossimo;
    }
    p->prossimo = ultimo;
    quanti = n;
}

Ruota::Ruota(const Ruota& t)
{
    quanti = t.quanti;
    Spicchio* q = t.ultimo;
    ultimo = new Spicchio(q->stato);
    Spicchio* p = ultimo;
    for (int i = 0; i < quanti - 1; i++) {
        q = q->prossimo;
        p->prossimo = new Spicchio(q->stato);
        p = p->prossimo;
    }
    p->prossimo = ultimo;
}

Ruota& Ruota::operator=(const Ruota& t)
{

```

```

    if (this != &t) {
        Spicchio* p = ultimo;
        Spicchio* q = ultimo->prossimo;
        for (int i = 0; i < quanti; i++) {
            delete p;
            p = q;
            q = q->prossimo;
        }
        quanti = t.quanti;
        q = t.ultimo;
        ultimo = new Spicchio(q->stato);
        p = ultimo;
        for (i = 0; i < quanti - 1; i++) {
            q = q->prossimo;
            p->prossimo = new Spicchio(q->stato);
            p = p->prossimo;
        }
        p->prossimo = ultimo;
    }
    return *this;
}

Ruota& Ruota::rendiOpaco(int s)
{
    (*this)[s % quanti].stato = OPACO;
    return *this;
}

Ruota& Ruota::rendiTrasparente(int s)
{
    (*this)[s % quanti].stato = TRASPARENTE;
    return *this;
}

int Ruota::seiOpaco(int s) const
{
    return (*this)[s % quanti].stato == OPACO;
}

Ruota& Ruota::gira(int p)
{
    for (int i = quanti - (p % quanti); i > 0; i--)
        ultimo = ultimo->prossimo;
    return *this;
}

Ruota& Ruota::sovrapponi(const Ruota& t)
{
    if (quanti != t.quanti)
        errore("Sovrapposizione di ruote di dimensioni diverse");
    else
    {
        Spicchio* p = ultimo;
        Spicchio* q = t.ultimo;
        for (int i = 0; i < quanti; i++) {
            p->stato = (p->stato == TRASPARENTE
                && q->stato == TRASPARENTE) ? TRASPARENTE : OPACO;
            p = p->prossimo;
            q = q->prossimo;
        }
    }
    return *this;
}

Ruota& Ruota::spicchi(int m)
{

```

```

    if (m > 0) {
        if (quanti > m)
            contrai(quanti - m);
        else
            if (quanti < m)
                espandi(m - quanti);
    }
    return *this;
}

Ruota::~Ruota()
{
    Spicchio* p = ultimo;
    Spicchio* q = ultimo->prossimo;
    for (int i = 0; i < quanti; i++) {
        delete p;
        p = q;
        q = q->prossimo;
    }
    quanti = 0;
}

ostream& operator<<(ostream& os, const Ruota& r)
{
    os << "<" << r.quanti << "> ";
    Ruota::Spicchio* q = r.ultimo->prossimo;
    for (int i = 0; i < r.quanti; i++) {
        os << (q->stato == Ruota::TRASPARENTE ? '-' : 'X');
        q = q->prossimo;
    }
    return os;
}

/*****
 *                               main.cp                               *
 *****/

#include <iostream.h>
#include "Ruota.h"

void main()
{
    const char Menu[] = "\nComandi disponibili:\n"
        "\tu (U) - a r1 (r2) viene assegnato r2 (r1)\n"
        "\to (O) - per rendere opaco uno spicchio di r1 (di r2)\n"
        "\tt (T) - per rendere trasparente uno spicchio di r1 (di r2)\n"
        "\tg (G) - per ruotare r1 (r2)\n"
        "\ts (S) - per sovrapporre r2 a r1 (r1 a r2)\n"
        "\tp (P) - per variare il numero di spicchi di r1 (di r2)\n"
        "\t{ (}) - per conoscere lo stato di uno spicchio di r1 (di r2)\n"
        "\t!   - per un esempio di concatenazione di operazioni su r1\n"
        "\t]   - esecuzione del costruttore di copia\n"
        "\t~   - per continuare con due nuove ruote\n"
        "\t'   - per terminare\n";

    char comando;
    do {
        cout << "Definisco due ruote r1 e r2.\n";
        cout << "Immetterne la dimensione: ";
        int n;
        cin >> n;
        Ruota r1(n);
        Ruota r2(r1);
        cout << "r1: " << r1 << "\nr2: " << r2 << '\n';
        cout << Menu << "\n\nComando? ";
    }

```

```
cin >> comando;
while (comando != '' && comando != '~')
{
    switch (comando) {
        case 'u': {
            r1 = r2;
            break;
        }
        case 'U': {
            r2 = r1;
            break;
        }
        case 'o': {
            int s;
            cin >> s;
            r1.rendiOpaco(s);
            break;
        }
        case 'O': {
            int s;
            cin >> s;
            r2.rendiOpaco(s);
            break;
        }
        case 't': {
            int s;
            cin >> s;
            r1.rendiTrasparente(s);
            break;
        }
        case 'T': {
            int s;
            cin >> s;
            r2.rendiTrasparente(s);
            break;
        }
        case 'g': {
            int s;
            cin >> s;
            r1.gira(s);
            break;
        }
        case 'G': {
            int s;
            cin >> s;
            r2.gira(s);
            break;
        }
        case 's': {
            int s;
            r1.sovrapponi(r2);
            break;
        }
        case 'S': {
            int s;
            r2.sovrapponi(r1);
            break;
        }
        case 'p': {
            int m;
            cin >> m;
            r1.spicchi(m);
            break;
        }
        case 'P': {
            int m;
```

```
        cin >> m;
        r2.spicchi(m);
        break;
    }
    case '{': {
        int s;
        cin >> s;
        cout << (r1.seiOpaco(s) ? "OPACO\n" : "TRASPARENTE\n");
        break;
    }
    case '}': {
        int s;
        cin >> s;
        cout << (r2.seiOpaco(s) ? "OPACO\n" : "TRASPARENTE\n");
        break;
    }
    case '!': {
        r1.rendiOpaco(1).rendiTrasparente(2).gira(1).spicchi(10);
        break;
    }
    case ']': {
        Ruota rr = r1;
        cout << "Nuova ruota: " << rr << '\n';
        break;
    }
    default:
        cout << "Comando non valido\n";
    }
    cout << "r1: " << r1 << "\nr2: " << r2 << '\n';
    cout << "Comando? ";
    cin >> comando;
}
} while (comando != '');
cout << "*\n";
}
```

Esercizio n. 2

Un `Percorso` è formato da caselle, ciascuna delle quali può assumere il colore bianco od il colore nero. Nel percorso transitano pedine bianche e pedine nere. Su ogni casella può sostare o transitare una sola pedina alla volta, e solo se il colore di tale pedina è uguale al colore di quella casella. Quando un pedina arriva sull'ultima casella, esce dal percorso. Le operazioni che possono essere effettuate su un percorso sono le seguenti:

- `Percorso p(N)`
Costruttore che inizializza un percorso `p` di lunghezza pari a `N` caselle. Inizialmente, tutte le caselle sono di colore bianco.
- `Percorso p(N, c)`
Costruttore che inizializza un percorso `p` di lunghezza pari a `N` caselle. Inizialmente, tutte le caselle sono di colore `c`.
- `Percorso p1(p)`
Costruttore di copia, che inizializza un percorso `p1` col valore del percorso `p`.
- `p1 = p`
Operatore di assegnamento, che sostituisce il valore del percorso risultato `p1` con quello del percorso `p`.
- `p.immetti(c)`
Operazione che causa l'immissione di una nuova pedina nel percorso `p`. Tale pedina ha colore `c`, e si porta nella casella più avanzata possibile. L'immissione ha successo solo se la prima casella del percorso `p` è vuota, ed ha colore `c`. Se una di queste condizioni non è verificata, l'operazione lascia il percorso inalterato.
- `p.cambiaColore(i)`
Operazione che cambia il colore della `i`-esima casella del percorso `p`, se tale casella è libera. Questa operazione può causare il movimento di una o più pedine. Ciascun pedina si porta nella casella più avanzata possibile.
- `~Percorso()`
Distruttore.
- `cout << p`
Operatore di uscita per il tipo `Percorso`. L'uscita ha la forma seguente:

```
bbBBnBBNnb
```

Il carattere 'b' rappresenta una casella libera bianca, il carattere 'n' rappresenta una casella libera nera, il carattere 'B' rappresenta una casella occupata da una pedina bianca, ed, infine, il carattere 'N' rappresenta una casella occupata da una pedina nera.

Utilizzando il linguaggio C++, realizzare il tipo di dati astratti `Percorso`, definito dalle precedenti specifiche, in modo tale che sia possibile concatenare le operazioni `immetti()` e `cambiaColore()`; esempio: `p.cambiaColore(i).immetti(c)`. Fare ricorso ad una rappresentazione del percorso ad array.

Soluzione

```

/*****
*
*                               Percorso.h
*****/

```

```

enum colore {BIANCO, NERO};
enum stato {LIBERA, OCCUPATA};

const int lunghezzaDiDefault = 10;

class Percorso {
    friend ostream& operator<<(ostream&, const Percorso&);
    class casella {
        colore col;
        stato st;
    public:
        casella(colore c = BIANCO) {col = c; st = LIBERA;}
        void cambiaColore() {col = (col == BIANCO) ? NERO : BIANCO;}
        void occupa() {st = OCCUPATA;}
        void libera() {st = LIBERA;}
        int seiOccupata() const {return st == OCCUPATA;}
        colore qualeColore() const {return col;}
    };
    casella *cc;
    int lunghezza;
    int avanza(int);
public:
    Percorso(int = lunghezzaDiDefault, colore = BIANCO);
    Percorso(const Percorso&);
    Percorso& operator=(const Percorso&);
    Percorso& immetti(colore);
    Percorso& cambiaColore(int);
    ~Percorso();
};

/*****
*
*                               Percorso.cp
*
*****/

#include <iostream.h>
#include <stdlib.h>
#include "Percorso.h"

void errore(const char* s)
{
    cerr << "Percorso: " << s << '\n';
    exit(1);
}

int Percorso::avanza(int i)
{
    for (int j = i + 1; j < lunghezza && !cc[j].seiOccupata() &&
        cc[i].qualeColore() == cc[j].qualeColore(); j++)
        ;
    return j - 1;
}

Percorso::Percorso(int lung, colore c)
{
    lunghezza = lung;
    cc = new casella[lunghezza];
    if (c == NERO)
        for (int i = 0; i < lunghezza; i++)
            cc[i].cambiaColore();
}

Percorso::Percorso(const Percorso& p)
{
    lunghezza = p.lunghezza;
    cc = new casella[lunghezza];
}

```

```

    for (int i = 0; i < lunghezza; i++)
        cc[i] = p.cc[i];
}

Percorso& Percorso::operator=(const Percorso& p)
{
    if (this != &p) {
        delete[] cc;
        lunghezza = p.lunghezza;
        cc = new casella[lunghezza];
        for (int i = 0; i < lunghezza; i++)
            cc[i] = p.cc[i];
    }
    return *this;
}

Percorso& Percorso::immetti(colore c)
{
    if (!cc[0].seiOccupata() && cc[0].qualeColore() == c) {
        int i = avanza(0);
        if (i < lunghezza - 1)
            cc[i].occupa();
    }
    return *this;
}

Percorso& Percorso::cambiaColore(int i)
{
    if (i < 1 || i > lunghezza) errore("Numero di casella errato");
    --i;
    if (!cc[i].seiOccupata()) {
        cc[i].cambiaColore();
        for (int j = i - 1; j >= 0 && cc[j].seiOccupata() &&
            cc[i].qualeColore() == cc[j].qualeColore(); j--) {
            cc[j].libera();
            int k = avanza(j);
            if (k < lunghezza - 1)
                cc[k].occupa();
        }
    }
    return *this;
}

Percorso::~~Percorso()
{
    delete[] cc;
}

ostream& operator<<(ostream& os, const Percorso& p)
{
    for (int i = 0; i < p.lunghezza; i++)
        if (p.cc[i].seiOccupata()) {
            if (p.cc[i].qualeColore() == BIANCO)
                os << 'B';
            else
                os << 'N';
        }
        else {
            if (p.cc[i].qualeColore() == BIANCO)
                os << 'b';
            else
                os << 'n';
        }
    return os;
}

```

```

/*****
*
*                               main.cp
*
*****/

#include <iostream.h>
#include "Percorso.h"

void main()
{
    const char Menu[] = "\nComandi disponibili:\n"
        "\tu (U) - a p1 (p2) viene assegnato p2 (p1)\n"
        "\tn (N) - per immettere una pedina nera in p1 (in p2)\n"
        "\tb (B) - per immettere una pedina bianca in p1 (in p2)\n"
        "\tc (C) - per cambiare il colore di una casella di p1 (di p2)\n"
        "\tp      - per un esempio di concatenazione di operazioni su p1\n"
        "\t~      - per continuare con due nuovi percorsi\n"
        "\t`      - per terminare\n";

    char comando;
do {
    cout << "Definisco due percorsi p1 e p2.\n";
    cout << "Immetterne la dimensione: ";
    int n;
    cin >> n;
    Percorso p1(n, NERO);
    Percorso p2(p1);
    cout << "p1: " << p1 << "\np2: " << p2 << '\n';
    cout << Menu << "\n\nComando? ";
    cin >> comando;
    while (comando != '' && comando != '~')
    {
        switch (comando) {
            case 'u': {
                p1 = p2;
                break;
            }
            case 'U': {
                p2 = p1;
                break;
            }
            case 'n': {
                p1.immetti(NERO);
                break;
            }
            case 'N': {
                p2.immetti(NERO);
                break;
            }
            case 'b': {
                p1.immetti(BIANCO);
                break;
            }
            case 'B': {
                p2.immetti(BIANCO);
                break;
            }
            case 'c': {
                int i;
                cin >> i;
                p1.cambiaColore(i);
                break;
            }
            case 'C': {
                int i;
                cin >> i;

```

```
        p2.cambiaColore(i);
        break;
    }
    case 'p': {
        int i;
        cin >> i;
        p1.cambiaColore(i).cambiaColore(i + 1).immetti(NERO);
        break;
    }
    default:
        cout << "Comando non valido\n";
    }
    cout << "p1: " << p1 << "\np2: " << p2 << '\n';
    cout << "Comando? ";
    cin >> comando;
}
} while (comando != '');
cout << "*\n";
}
```

Esercizio n. 3

Un Contenitore è in grado di contenere elementi in numero limitato (la *capienza* del contenitore). Ciascun elemento ha un nome che consiste in una lettera dell'alfabeto. Più elementi di un contenitore possono avere lo stesso nome. Le operazioni che possono essere effettuate su un contenitore sono le seguenti:

- Contenitore `c()`
Costruttore di default, che inizializza un contenitore `c` avente capienza pari a 260 elementi. Tale contenitore è vuoto.
- Contenitore `c(N)`
Costruttore che inizializza un contenitore `c` avente capienza pari ad `N` elementi. Tale contenitore è vuoto.
- Contenitore `c1(c)`
Costruttore di copia, che inizializza un contenitore `c1` col valore del contenitore `c`.
- `c1 = c`
Operatore di assegnamento, che sostituisce il valore del contenitore risultato `c1` con quello del contenitore `c`. La capienza di `c1` è resa pari a quella di `c`.
- `c += p`
Operatore di assegnamento composto, che inserisce nel contenitore `c` un elemento avente nome `p`.
- `c -= p`
Operatore di assegnamento composto, che estrae dal contenitore `c` *tutti* gli elementi aventi nome `p`.
- `c % N`
Operatore modulo, che ritorna un contenitore avente capienza `N` e contenuto identico a quello del contenitore `c`. Il numero di elementi contenuti in `c` deve essere minore od uguale ad `N`.
- `c1 * c2`
Operatore di moltiplicazione, che ritorna un contenitore con un elemento per ciascuno dei nomi di elemento diversi presenti in entrambi i contenitori `c1` e `c2`. La capienza del contenitore risultato è pari alla minore tra le capienze di `c1` e `c2`.
- `c1 / c2`
Operatore di divisione, che ritorna un contenitore con un elemento per ciascuno dei nomi di elemento diversi presenti nel contenitore `c1` che non sono presenti nel contenitore `c2`. La capienza del contenitore risultato è pari alla capienza di `c1`.
- `c.capienza()`
Operazione che ritorna la capienza del contenitore `c`.
- `c.quantità()`
Operazione che ritorna il numero degli elementi presenti nel contenitore `c`.
- `~Contenitore()`
Distruttore.
- `cout << c`
Operatore di uscita per il tipo Contenitore. L'uscita ha la forma seguente:
`<a:3, d:1, s:5>`
In questo esempio, nel contenitore `c` sono presenti tre elementi di nome 'a', uno di nome 'd' e cinque di nome 's'.

Utilizzando il linguaggio C++, realizzare il tipo di dati astratti Contenitore, definito dalle precedenti specifiche.

Soluzione

```

/*****
 *                               Contenitore.h                               *
 *****/

class Contenitore {
    friend Contenitore operator*(const Contenitore, const Contenitore);
    friend Contenitore operator/(const Contenitore, const Contenitore);
    friend ostream& operator<<(ostream&, const Contenitore&);
    int numElem, diversi, cap;
    int ee[26];
public:
    Contenitore(int = 260);
    // Contenitore(const Contenitore&);
    // Contenitore& operator=(const Contenitore&);
    Contenitore& operator+=(char);
    Contenitore& operator-=(char);
    Contenitore operator%(int);
    int capienza() const {return cap;}
    int quanti() const {return numElem;}
    // ~Contenitore();
};

/*****
 *                               Contenitore.cp                               *
 *****/

#include <iostream.h>
#include <stdlib.h>
#include "Contenitore.h"

void errore(const char* s)
{
    cerr << "Contenitore: " << s << '\n';
    exit(1);
}

Contenitore::Contenitore(int N)
{
    for (int i = 0; i <= 25; i++)
        ee[i] = 0;
    numElem = diversi = 0;
    cap = N;
}

Contenitore& Contenitore::operator+=(char p)
{
    if (numElem >= cap)
        errore("Capienza insufficiente");
    if (p < 'a' || p > 'z')
        errore("Nome di elemento errato");
    if (!ee[p - 'a']++)
        diversi++;
    numElem++;
    return *this;
}

Contenitore& Contenitore::operator-=(char p)
{

```

```

    if (p < 'a' || p > 'z')
        errore("Nome di elemento errato");
    int i = p - 'a';
    if (ee[i]) {
        numElem = numElem - ee[i];
        diversi--;
        ee[i] = 0;
    }
    return *this;
}

Contenitore Contenitore::operator%(int N)
{
    if (numElem > N)
        errore ("Capienza insufficiente");
    Contenitore c(*this);
    c.cap = N;
    return c;
}

Contenitore operator*(const Contenitore c1, const Contenitore c2)
{
    Contenitore c((c1.cap >= c2.cap) ? c2.cap : c1.cap);
    for (int i = 0; i <= 25; i++) {
        if (c1.ee[i] > 0 && c2.ee[i] > 0) {
            c.ee[i] = 1;
            c.numElem++;
        }
        c.diversi = c.numElem;
    }
    return c;
}

Contenitore operator/(const Contenitore c1, const Contenitore c2)
{
    Contenitore c(c1.cap + c2.cap);
    for (int i = 0; i <= 25; i++) {
        if (c1.ee[i] > 0 && c2.ee[i] == 0) {
            c.ee[i] = 1;
            c.numElem++;
        }
        c.diversi = c.numElem;
    }
    return c;
}

ostream& operator<<(ostream& os, const Contenitore& c)
{
    int i = 0;
    os << '<';
    while (i < 25 && c.ee[i] == 0)
        i++;
    if (c.ee[i] > 0) {
        os << char(i + 'a') << ':' << c.ee[i];
        while (++i <= 25)
            if (c.ee[i])
                os << ", " << char(i + 'a') << ':' << c.ee[i];
    }
    os << ">";
    return os;
}

/*****
*
*                               main.cp
*
*****/

```

```

#include <iostream.h>
#include "Contenitore.h"

void main()
{
    const char Menu[] = "\nComandi disponibili:\n"
        "\tu (U) - a c1 (c2) viene assegnato c2 (c1)\n"
        "\ti (I) - per inserire un elemento in c1 (in c2)\n"
        "\te (E) - per estrarre elementi da c1 (da c2)\n"
        "\tc (C) - per modificare la capienza di c1 (di c2)\n"
        "\tm (M) - a c1 (c2) viene assegnato c1 * c2\n"
        "\td (D) - a c1 (c2) viene assegnato c1 / c2\n"
        "\tf      - per informazioni su c1 e c2\n"
        "\t~      - per continuare con due nuovi contenitori\n"
        "\t'      - per terminare\n";

    char comando;
    do {
        cout << "Definisco due contenitori c1 e c2.\n";
        cout << "Immettere la dimensione: ";
        int n;
        cin >> n;
        Contenitore c1(n);
        Contenitore c2(c1);
        cout << "c1: " << c1 << "\nc2: " << c2 << '\n';
        cout << Menu << "\n\nComando? ";
        cin >> comando;
        while (comando != '' && comando != '~')
        {
            switch (comando) {
                case 'u': {
                    c1 = c2;
                    break;
                }
                case 'U': {
                    c2 = c1;
                    break;
                }
                case 'i': {
                    char p;
                    cin >> p;
                    c1 += p;
                    break;
                }
                case 'I': {
                    char p;
                    cin >> p;
                    c2 += p;
                    break;
                }
                case 'e': {
                    char p;
                    cin >> p;
                    c1 -= p;
                    break;
                }
                case 'E': {
                    char p;
                    cin >> p;
                    c2 -= p;
                    break;
                }
                case 'c': {
                    int i;
                    cin >> i;
                }
            }
        }
    }
}

```

```
        c1 = c1 % i;
        break;
    }
    case 'C': {
        int i;
        cin >> i;
        c2 = c2 % i;
        break;
    }
    case 'm': {
        c1 = c1 * c2;
        break;
    }
    case 'M': {
        c2 = c1 * c2;
        break;
    }
    case 'd': {
        c1 = c1 / c2;
        break;
    }
    case 'D': {
        c2 = c2 / c1;
        break;
    }
    case 'f': {
        cout << "Capienza di c1: " << c1.capienza() << '\t'
              << "Elementi in c1: " << c1.quant() << '\n';
        cout << "Capienza di c2: " << c2.capienza() << '\t'
              << "Elementi in c2: " << c2.quant() << '\n';
        break;
    }
    default: {
        cout << "Comando non valido\n";
    }
}
cout << "c1: " << c1 << "\nc2: " << c2 << '\n';
cout << "Comando? ";
cin >> comando;
}
} while (comando != '');
cout << "*\n";
}
```

Esercizio n. 4

Una *lista di interi* è formata da elementi a valore intero. Il peso di una lista di interi è dato dal numero di elementi che la formano. Una *lista multipla* è formata da al più T liste di interi. Le operazioni che possono essere effettuate su una lista multipla sono le seguenti:

- `ListaMultipla m()`
Costruttore di default, che inizializza una lista multipla m . Inizialmente, tale lista multipla è vuota.
- `ListaMultipla m(n)`
Costruttore che inizializza una lista multipla formata da n liste di interi. Inizialmente, tali liste di interi sono vuote.
- `ListaMultipla m1(m)`
Costruttore di copia, che inizializza una lista multipla $m1$ col valore della lista multipla m .
- `m1 = m`
Operatore di assegnamento, che sostituisce il valore della lista multipla risultato $m1$ con quello della lista multipla m .
- `m.inserisci(i)`
Operazione che inserisce un elemento a valore intero i nella lista di interi a minor peso della lista multipla m .
- `m.estrai()`
Operazione che estrae un elemento a valore intero dalla lista di interi a maggior peso della lista multipla m , e ritorna il valore di tale elemento.
- `m += n`
Operatore di somma e assegnamento, che aggiunge n liste di interi alla lista multipla m . Tali liste di interi sono vuote.
- `m -= n`
Operatore di sottrazione e assegnamento, che elimina dalla lista multipla m le n liste di interi aventi peso maggiore.
- `~ListaMultipla()`
Distruttore.
- `cout << m`
Operatore di uscita per il tipo `ListaMultipla`. L'uscita ha la forma seguente:

```
[15] [2, 19] [1, 2, 1] [2, 4, 3]
```

Le parentesi quadre indicano una lista di interi. Le liste di interi vengono scritte secondo pesi crescenti. In questo esempio, la lista multipla è formata da quattro liste di interi; la prima di tali liste di interi è formata da un solo elemento avente valore 15.

Utilizzando il linguaggio C++, realizzare il tipo di dati astratti `ListaMultipla` definito dalle precedenti specifiche. Individuare eventuali situazioni di errore, e metterne in opera un corretto trattamento.

Soluzione (a cura di F. Potortì)

```
/*
 *
 *                               lm.h
 *
 */
```

```

const int T = 10;

class ListaMultipla {
public:
    friend ostream& operator<<(ostream&, const ListaMultipla&);
    ListaMultipla(const int = 0);
    ListaMultipla(const ListaMultipla&);
    ListaMultipla& operator=(const ListaMultipla&);
    ListaMultipla& inserisci(const int);
    int estrai();
    ListaMultipla& operator+=(const int);
    ListaMultipla& operator-=(const int);
    ~ListaMultipla();
private:
    struct elem {
        elem *next;
        int value;
    };
    struct capolista {
        elem *first;
        int peso;
    };
    capolista lista [T];
    int nliste;
    void inicializza(const int n);
    void copia(const ListaMultipla& m);
    void distruggi();
    void ordina();
};

/*****
*
*                               lm.cp
*
*****/

#include <iostream.h>
#include <stdlib.h>
#include "lm.h"

void errore(char *messaggio)
{
    cerr << "ListaMultipla: " << messaggio << endl;
    exit(1);
}

void ListaMultipla::ordina()
{
    for (int i = 1; i < nliste; i++) {
        capolista cl = lista[i];
        int p = cl.peso;
        int j = i - 1;
        while (p < lista[j].peso) {
            lista[j + 1] = lista[j];
            if (--j < 0) break;
        }
        lista[j + 1] = cl;
    }
}

void ListaMultipla::inicializza(const int n)
{
    for (int i = 0; i < n; i++) {
        lista[i].peso = 0;
        lista[i].first = NULL;
    }
    nliste = n;
}

```

```

}

ListaMultipla::ListaMultipla(const int n)
{
    if (n < 0)
        errore("la dimensione della lista multipla e' negativa");
    if (n > T)
        errore("la dimensione della lista multipla eccede il massimo");
    inizializza(n);
}

void ListaMultipla::distruggi()
{
    for (int i = 0; i < nliste; i++)
        while (lista[i].first != NULL) {
            elem *e = lista[i].first->next;
            delete lista[i].first;
            lista[i].first = e;
        }
}

ListaMultipla::~ListaMultipla()
{
    distruggi();
}

void ListaMultipla::copia(const ListaMultipla& m)
{
    inizializza(m.nliste);
    for (int i = 0; i < nliste; i++) {
        for (elem *me = m.lista[i].first; me != NULL; me = me->next) {
            elem *e = new elem;
            e->value = me->value;
            e->next = lista[i].first;
            lista[i].first = e;
        }
        lista[i].peso = m.lista[i].peso;
    }
}

ListaMultipla::ListaMultipla(const ListaMultipla& m)
{
    copia(m);
}

ListaMultipla& ListaMultipla::operator=(const ListaMultipla& m)
{
    if (this != &m) {
        distruggi();
        copia(m);
    }
    return *this;
}

ListaMultipla& ListaMultipla::inserisci(const int i)
{
    if (nliste == 0)
        errore("non ci sono liste");
    elem *e = new elem;
    e->value = i;
    e->next = lista[0].first;
    lista[0].first = e;
    lista[0].peso += i;
    ordina();
    return *this;
}

```

```

int ListaMultipla::estrai()
{
    if (nliste == 0)
        errore("non ci sono liste");
    elem *e = lista[nliste - 1].first;
    if (e == NULL)
        errore("la lista piu' leggera e' vuota");
    int i = e->value;
    lista[nliste - 1].first = e->next;
    delete e;
    lista[nliste - 1].peso -= i;
    ordina();
    return i;
}

ListaMultipla& ListaMultipla::operator+=(const int n)
{
    if (n < 0)
        errore("il numero di liste da aggiungere e' negativo");
    if (nliste + n > T)
        errore("la dimensione della lista multipla eccede il massimo");
    for (int i = nliste; i < nliste + n; i++) {
        lista[i].peso = 0;
        lista[i].first = NULL;
    }
    nliste += n;
    ordina();
    return *this;
}

ListaMultipla& ListaMultipla::operator-=(const int n)
{
    if (n < 0)
        errore("il numero di liste da togliere non puo' essere negativo");
    if (n > nliste)
        errore("la dimensione della lista multipla diventa negativa");
    for (int i = nliste - n; i < nliste; i++)
        while (lista[i].first != NULL) {
            elem *e = lista[i].first->next;
            delete lista[i].first;
            lista[i].first = e;
        }
    nliste -= n;
    return *this;
}

ostream& operator<<(ostream& os, const ListaMultipla& m)
{
    for (int i = 0; i < m.nliste; i++) {
        os << "[";
        for (ListaMultipla::elem *e = m.lista[i].first; e != NULL; e = e->next) {
            os << e->value;
            if (e->next != NULL)
                os << ", ";
        }
        os << "]";
        if (i < m.nliste-1)
            os << " ";
    }
    return os;
}

```

Esercizio n. 5

Un `Identificatore` è formato da una lettera maiuscola o minuscola dell'alfabeto e da un numero intero compreso tra 0 e 9. Le operazioni che possono essere effettuate su un identificatore sono le seguenti:

- `Identificatore d()`
Costruttore di default, che inizializza un identificatore `d`. Tale identificatore è formato dalla lettera 'A' e dall'intero 0.
- `Identificatore d(c, n)`
Costruttore che inizializza un identificatore `d`. Tale identificatore è formato dalla lettera `c` e dall'intero `n`.
- `Identificatore d(n)`
Costruttore di conversione, che consente di usare un `int` ovunque occorre un identificatore. Il costruttore inizializza l'identificatore `d` con la lettera 'A' e l'intero `n`.
- `Identificatore d1(d)`
Costruttore di copia, che inizializza un identificatore `d1` col valore dell'identificatore `d`.
- `operator int(d)`
Operatore di conversione, che consente di usare un identificatore ovunque occorre un `int`. L'operatore ritorna il numero dell'identificatore `d`.
- `operator char(d)`
Operatore di conversione, che consente di usare un identificatore ovunque occorre un `char`. L'operatore ritorna la lettera dell'identificatore `d`.
- `d1 = d`
Operatore di assegnamento, che sostituisce il valore dell'identificatore risultato `d1` con quello dell'identificatore `d`.
- `cout << d`
Operatore di uscita per il tipo `Identificatore`. L'uscita consiste nella lettera e nel numero che formano l'identificatore `d`. Esempio:
 C3
- `cin >> d`
Operatore di ingresso per il tipo `Identificatore`. L'operatore legge il risultato dell'operatore di uscita per il tipo `Identificatore`.
- `~Identificatore()`
Distruttore.

Un `Recipiente` è in grado di contenere un numero illimitato di identificatori. Le operazioni che possono essere effettuate su un recipiente sono le seguenti:

- `Recipiente s()`
Costruttore di default, che inizializza un recipiente `s`. Inizialmente, il recipiente non contiene identificatori.
- `Recipiente s1(s)`
Costruttore di copia, che inizializza un recipiente `s1` col valore del recipiente `s`.
- `s1 = s`
Operatore di assegnamento, che sostituisce il valore del recipiente risultato `s1` con quello del reci-

piante s.

- s.inserisci(d)
Operazione che inserisce l'identificatore d nel recipiente s.
- s.quantit(d)
Operazione che ritorna il numero di identificatori presenti nel recipiente s ed aventi valore d.
- s.estrai(d)
Operazione che estrae dal recipiente s tutti gli identificatori aventi valore d.
- ~Recipiente()
Distruttore.

Mediante il linguaggio C++, realizzare il tipo di dati astratti *Identificatore*, definito dalle precedenti specifiche. Utilizzare il tipo *Identificatore* per realizzare il tipo di dati astratti *Recipiente* in modo tale che sia possibile concatenare le operazioni *inserisci()* e *quantit()*, e le operazioni *estrai()* e *quantit()*; esempio *s.inserisci(d).quantit(d)*. Individuare le eventuali situazioni di errore, e metterne in opera un corretto trattamento.

Soluzione (a cura di C. Bernardeschi)

```
#include <iostream.h>
#include <stdlib.h>
#include <ctype.h>

class Identificatore {
    friend ostream& operator<<(ostream&, const Identificatore&);
    friend istream& operator>>(istream&, Identificatore&);
    char lett;
    int num;
public:
    Identificatore(char c = 'A', int n = 0)
    {
        if ((islower(c) || isupper(c)) && (0 <= n && n <= 9)) {
            lett = c;
            num = n;
        }
        else {
            lett = 'A';
            num = 0;
        }
    }
    Identificatore(int n)
    {
        lett = 'A';
        if (0 <= n && n <= 9)
            num = n;
        else
            num = 0;
    }
    // Identificatore(const Identificatore&);
    operator int() const
    {
        return num;
    }
    operator char() const
    {
        return lett;
    }
    // Identificatore& operator=(const Identificatore&);
    // ~Identificatore();
};
```

```

ostream& operator<<(ostream& os, const Identificatore& d) {
    char c = '0' + num;
    os << d.lett << c;
    return os;
}

istream& operator>>(istream& is, Identificatore& d) {
    char c;
    if (!(is >> c))
        return is;
    if (!(islower(c) || isupper(c))) {
        is.clear(ios::failbit);
        return is;
    }
    char c1;
    if (!(is.get(c1)))
        return is;
    int i = c1 - '0';
    if (!(0 <= i && i <= 9)) {
        is.clear(ios::failbit);
        return is;
    }
    d = Identificatore(c, i);
    return is;
}

class Recipiente {
    int vett[520];
    int indice(char c, int j)
    {
        if (islower(c))
            return (c - 'a') * 10 + j;
        else
            return 260 + (c - 'A') * 10 + j;
    }
public:
    Recipiente()
    {
        for (int i = 0; i < 520; i++)
            vett[i] = 0;
    }
    // Recipiente(const Recipiente&);
    // Recipiente& operator=(const Recipiente&);
    Recipiente& inserisci(const Identificatore& d)
    {
        vett[indice(char(d), int(d))]+=;
        return *this;
    }
    int quanti(const Identificatore& d)
    {
        return vett[indice(char(d), int(d))];
    }
    Recipiente& estrai(const Identificatore& d)
    {
        vett[indice(char(d), int(d))] = 0; return *this;
    }
    // ~Recipiente();
};

```

Esercizio n. 6

Una *Striscia* è formata da caselle trasparenti o colorate, ed i possibili colori sono BIANCO e NERO. Le caselle sono numerate a partire da 1. Le caselle bianche occupano *sempre* le posizioni della striscia ai più bassi numeri d'ordine, le caselle nere occupano *sempre* le posizioni della striscia ai più alti numeri d'ordine. Le operazioni che possono essere effettuate su una striscia sono le seguenti:

- `Striscia s()`
Costruttore di default, che inizializza una striscia `s` formata da una sola casella. Inizialmente, tale casella è trasparente.
- `Striscia s(n)`
Costruttore che inizializza una striscia `s` formata da `n` caselle. Inizialmente, tali caselle sono trasparenti.
- `Striscia s1(s)`
Costruttore di copia, che inizializza una striscia `s1` col valore della striscia `s`.
- `s1 = s`
Operatore di assegnamento, che sostituisce il valore della striscia risultato `s1` con quello della striscia `s`.
- `s += c`
Operatore di somma e assegnamento, che colora una casella trasparente della striscia `s`. Tale casella assume il colore `c`.
- `s + c`
Operatore di somma, che ritorna la striscia ottenuta facendo assumere il colore `c` ad una casella trasparente della striscia `s`.
- `s -= c`
Operatore di sottrazione e assegnamento, che rende trasparente una casella di colore `c` della striscia `s`.
- `s - c`
Operatore di sottrazione, che ritorna la striscia ottenuta rendendo trasparente una casella di colore `c` della striscia `s`.
- `s *= n`
Operatore di prodotto e assegnamento, che amplia la striscia `s` aggiungendo ad essa `n` caselle trasparenti.
- `s /= n`
Operatore di divisione e assegnamento, che riduce la striscia `s` eliminando da essa `n` caselle trasparenti.
- `s % c`
Operatore di modulo, che ritorna il numero di caselle di colore `c` della striscia `s`.
- `~Striscia()`
Distruttore.
- `cout << s`
Operatore di uscita per il tipo `Striscia`. L'uscita ha la forma seguente:
[BBB^^^NNNNN]
I caratteri 'B' ed 'N' rappresentano caselle di colore BIANCO e NERO, rispettivamente, il carattere '^'

rappresenta una casella trasparente.

- `cin >> s`

Operatore di ingresso per il tipo `Striscia`. L'operatore legge il risultato dell'operatore di uscita per il tipo `Striscia`.

Utilizzando il linguaggio C++, realizzare il tipo di dati astratti `Striscia`, definito dalle precedenti specifiche. Si individuino le eventuali situazioni di errore, e se ne metta in opera un corretto trattamento.

Soluzione (a cura di F. Potortì)

```
#include <iostream.h>

enum Colore {BIANCO, NERO};

class Striscia {
    friend ostream& operator<<(ostream&, const Striscia&);
    friend istream& operator>>(istream&, Striscia&);
    int ccol[2], ctra;
public:
    Striscia(int n = 1)
    {
        ccol[BIANCO] = ccol[NERO] = 0;
        if (n > 0) ctra = n;
        else ctra = 0;
    }
    Striscia& operator+=(Colore c)
    {
        if (ctra > 0) {
            ctra -= 1;
            ccol[c] += 1;
        }
        return *this;
    }
    Striscia operator+(Colore c) const
    {
        Striscia retval(*this);
        return (retval += c);
    }
    Striscia& operator--=(Colore c)
    {
        if (ccol[c] > 0) {
            ccol[c] -= 1;
            ctra += 1;
        }
        return *this;
    }
    Striscia operator-(Colore c) const
    {
        Striscia retval(*this);
        return (retval -= c);
    }
    Striscia& operator*=(int n)
    {
        if (n > 0)
            ctra += n;
        return *this;
    }
    Striscia& operator/=(int n)
    {
        if (n > 0)
            if ((ctra -= n) < 0) ctra = 0;
        return *this;
    }
};
```

```

    }
    int operator%(Colore c) const
    {
        return ccol[c];
    }
};

ostream& operator<<(ostream& os, const Striscia& s)
{
    int i;
    os << '[';
    for (i = 0; i < s.ccol[BIANCO]; i++)
        os << 'B';
    for (i = 0; i < s.ctra; i++)
        os << '^';
    for (i = 0; i < s.ccol[NERO]; i++)
        os << 'N';
    os << ']';
    return os;
}

istream& operator>>(istream& is, Striscia& s)
{
    char c;
    int stato = 1;
    Striscia tmp(0);
    if (!(is >> c))
        return is;
    if (c != '[') {
        is.clear(ios::failbit);
        return is;
    }
    while (is.get(c))
        switch (c) {
            case 'B':
                if (stato > 1) {
                    is.clear(ios::failbit);
                    return is;
                }
                tmp.ccol[BIANCO] += 1;
                break;
            case '^':
                if (stato > 2) {
                    is.clear(ios::failbit);
                    return is;
                }
                stato = 2;
                tmp.ctra += 1;
                break;
            case 'N':
                stato = 3;
                tmp.ccol[NERO] += 1;
                break;
            case ']':
                s = tmp;
                return is;
            default:
                is.clear(ios::failbit);
                return is;
        }
    return is;
}

```

Esercizio n. 7

Un `Dado` ha valore intero compreso tra 1 e 6. Le operazioni che possono essere effettuate su un dado sono le seguenti:

- `Dado d()`
Costruttore di default, che inizializza un dado `d`. Inizialmente, tale dado ha valore 1.
- `Dado d1(d)`
Costruttore di copia, che inizializza un dado `d1` col valore del dado `d`.
- `Dado d(n)`
Costruttore di conversione, che consente di usare un `int` ovunque occorre un dado. Il costruttore inizializza un dado `d` col valore intero `n`.
- `operator int(d)`
Operatore di conversione, che consente di usare un dado ovunque occorre un `int`. L'operatore ritorna il valore del dado `d`.
- `d1 = d`
Operatore di assegnamento, che sostituisce il valore del dado risultato `d1` con quello del dado `d`.
- `~Dado()`
Distruttore.

Un `LancioDiDadi` è in grado di contenere un numero limitato di dadi. Le operazioni che possono essere effettuate su un lancio di dadi sono le seguenti:

- `LancioDiDadi c()`
Costruttore di default, che inizializza un lancio di dadi `c`. Tale lancio di dadi può contenere al più un numero di dadi pari al valore dell'intero costante `M`. Inizialmente, il lancio di dadi non contiene dadi.
- `LancioDiDadi c1(c)`
Costruttore di copia, che inizializza un lancio di dadi `c1` col valore del lancio di dadi `c`.
- `LancioDiDadi c(n)`
Costruttore di conversione, che consente di usare un `int` ovunque occorre un lancio di dadi. Il costruttore inizializza un lancio di dadi `c`. Tale lancio di dadi può contenere al più `n` dadi. Inizialmente, il lancio di dadi non contiene dadi.
- `operator int(c)`
Operatore di conversione, che consente di usare un lancio di dadi ovunque occorre un `int`. L'operatore ritorna la somma dei valori dei dadi che formano il lancio di dadi `c`.
- `c1 = c`
Operatore di assegnamento, che sostituisce il valore del lancio di dadi risultato `c1` con quello del lancio di dadi `c`.
- `c += d`
Operatore di somma e assegnamento, che aggiunge il dado `d` al lancio di dadi `c`.
- `c -= d`
Operatore di sottrazione e assegnamento, che elimina il dado `d` dal lancio di dadi `c`.
- `c %= d`
Operatore di modulo e assegnamento, che elimina dal lancio di dadi `c` tutti i dadi aventi valore pari al dado `d`.

- !c
Operatore di negazione logica, che ritorna il numero di dadi contenuti nel lancio di dadi c.
- cout << c
Operatore di uscita per il tipo LancioDiDadi. L'uscita consiste nei valori dei dadi che formano il lancio c, in ordine crescente, separati da virgole e racchiusi tra parentesi graffe. Esempio:
{1, 1, 1, 4, 5}
In questo esempio, il lancio è formato da cinque dadi, ed i primi tre hanno valore 1.
- ~LancioDiDadi()
Distruttore.

Mediante il linguaggio C++, realizzare il tipo di dati astratti Dado, definito dalle precedenti specifiche. Utilizzare il tipo Dado per realizzare il tipo di dati astratti LancioDiDadi. Individuare le eventuali situazioni di errore, e metterne in opera un corretto trattamento.

Soluzione (a cura di F. Potortì)

```
#include <iostream.h>

enum {FACCE = 6};

class Dado {
    int faccia;
public:
    Dado(int n = 1)
    {
        faccia = 1 + (n - 1) % FACCE;
    }
    // Dado(const Dado&);
    operator int() const
    {
        return faccia;
    }
    // Dado& operator=(const Dado&);
    // ~Dado();
};

class LancioDiDadi {
    friend ostream& operator<<(ostream&, const LancioDiDadi&);
    static const int M;
    int dadi[FACCE];
    int maxdadi;
    int ndadi;
    int lancio;
public:
    LancioDiDadi(unsigned int n = M) : maxdadi(n), ndadi(0), lancio(0)
    {
        if (n < 0)
            maxdadi = 0;
        for (int i = 0; i < FACCE; i++)
            dadi[i] = 0;
    }
    // LancioDiDadi(const LancioDiDadi&);
    operator int() const
    {
        return lancio;
    }
    // LancioDiDadi& operator=(const LancioDiDadi&);
    LancioDiDadi& operator+=(const Dado& d)
    {
        if (ndadi < maxdadi) {
```

```

        ++dadi[d];
        ++ndadi;
        lancio += d;
    }
    return *this;
}
LancioDiDadi& operator--=(const Dado& d)
{
    if (dadi[d] > 0) {
        --dadi[d];
        --ndadi;
        lancio -= d;
    }
    return *this;
}
LancioDiDadi& operator%=(const Dado& d)
{
    lancio -= int(d) * dadi[d];
    ndadi -= dadi[d];
    dadi[d] = 0;
    return *this;
}
int operator!() const
{
    return ndadi;
}
// ~LancioDiDadi();
};

const int LancioDiDadi::M = 10;

ostream& operator<<(ostream& os, const LancioDiDadi& l)
{
    os << '{';
    int primo = 1;
    for (int f = 0; f < FACCE; f++)
        for (int i = 0; i < l.dadi[f]; i++) {
            if (primo)
                primo = 0;
            else
                os << ", ";
            os << f;
        }
    os << '}';
    return os;
}

```

Esercizio n. 8

Un `InsiemeDiInteri` è formato da interi *diversi* compresi tra 0 ed $N-1$, con N minore o uguale al numero di bit che formano un `long int`. Le operazioni che possono essere effettuate su un insieme di interi sono le seguenti:

- `InsiemeDiInteri d()`
Costruttore di default, che inizializza un insieme di interi `d`. Inizialmente, tale insieme di interi è vuoto.
- `InsiemeDiInteri d1(d)`
Costruttore di copia, che inizializza un insieme di interi `d1` col valore dell'insieme di interi `d`.
- `operator int(d)`
Operatore di conversione, che consente di usare un insieme di interi ovunque occorre un `int`. L'operatore ritorna la somma degli interi che formano l'insieme di interi `d`.
- `d1 = d`
Operatore di assegnamento, che sostituisce il valore dell'insieme di interi risultato `d1` con quello dell'insieme di interi `d`.
- `d += i`
Operatore di somma e assegnamento, che inserisce l'intero `i` nell'insieme di interi `d`.
- `d -= i`
Operatore di sottrazione e assegnamento, che elimina l'intero `i` dall'insieme di interi `d`.
- `cout << d`
Operatore di uscita per il tipo `InsiemeDiInteri`. L'uscita consiste nei valori degli interi che formano l'insieme di interi `d`, in ordine crescente, separati da virgole e racchiusi tra parentesi graffe.
Esempio:
 {1, 3, 8, 14}
- `~InsiemeDiInteri()`
Distruttore.

Un `Collezione` è formata da insiemi di interi in numero limitato (la *capienza* della collezione). Le operazioni che possono essere effettuate su una collezione sono le seguenti:

- `Collezione c()`
Costruttore di default, che inizializza una collezione `c` avente capienza pari al valore dell'intero costante M . Inizialmente, tale collezione è vuota.
- `Collezione c(m)`
Costruttore che inizializza una collezione `c` avente capienza `m`. Inizialmente, tale collezione è vuota.
- `Collezione c1(c)`
Costruttore di copia, che inizializza una collezione `c1` col valore della collezione `c`.
- `operator int(c)`
Operatore di conversione, che consente di usare una collezione ovunque occorre un `int`. L'operatore ritorna la somma degli interi che formano gli insiemi di interi contenuti in `c`.
- `c1 = c`
Operatore di assegnamento, che sostituisce il valore della collezione risultato `c1` con quello della collezione `c`.

- `c += d`
Operatore di somma e assegnamento, che inserisce l'insieme di interi `d` nella collezione `c`.
- `c -= d`
Operatore di sottrazione e assegnamento, che elimina l'insieme di interi `d` dalla collezione `c`.
- `cout << c`
Operatore di uscita per il tipo `Collezione`. L'uscita consiste nei valori degli insiemi di interi che formano la collezione `c`. Esempio:

```
{1, 3, 8, 14} {1} {0, 3, 5}
```

 In questo esempio, il secondo insieme della collezione è formato da un solo intero avente valore 1.
- `~Collezione()`
Distruttore.

Mediante il linguaggio C++, realizzare il tipo di dati astratti `InsiemeDiInteri`, definito dalle precedenti specifiche. Utilizzare il tipo `InsiemeDiInteri` per realizzare il tipo di dati astratti `Collezione`. Individuare le eventuali situazioni di errore, e metterne in opera un corretto trattamento.

Soluzione (a cura di F. Potortì)

```
#include <iostream.h>
#include <stdlib.h>

class InsiemeDiInteri {
    friend ostream& operator<<(ostream&, InsiemeDiInteri);
    enum {N = 8 * sizeof(long int)};
    long int insieme;
    int somma;
public:
    InsiemeDiInteri()
    {
        insieme = somma = 0;
    }
    operator int() const
    {
        return somma;
    }
    InsiemeDiInteri& operator+=(int i)
    {
        if (i >= 0 && i < N && (insieme & 1L << i) == 0) {
            insieme |= 1L << i;
            somma += i;
        }
        return *this;
    }
    InsiemeDiInteri& operator-=(int i)
    {
        if (i >= 0 && i < N && (~insieme & 1L << i) == 0) {
            insieme &= ~(1L << i);
            somma -= i;
        }
        return *this;
    }
    int operator==(InsiemeDiInteri idi) const {
        return (insieme == idi.insieme);
    }
};

ostream& operator<<(ostream& os, InsiemeDiInteri idi)
{
    os << '{';
```

```

    int i = 0;
    unsigned long int k = idi.insieme;
    for (;;) {
        if (k & 1)
            os << i;
        if ((k >>= 1) == 0)
            break;
        else
            os << ", ";
    }
    os << '}'';
    return os;
}

class Collezione {
    static const int M;
    int size;
    int contents;
    InsiemeDiInteri *c;
public:
    Collezione(int n = M)
    {
        size = contents = 0;
        if (n > 0)
            c = new InsiemeDiInteri[size = n];
    }
    Collezione(const Collezione& cc);
    operator int() const;
    Collezione& operator=(const Collezione& cc);
    Collezione& operator+=(InsiemeDiInteri idi)
    {
        if (contents < size)
            c[contents++] = idi;
        return *this;
    }
    Collezione& operator--=(InsiemeDiInteri idi);
    friend ostream& operator<<(ostream&, const Collezione&);
    ~Collezione()
    {
        if (size > 0)
            delete [] c;
    }
};

const int Collezione::M = 10;

Collezione::Collezione(const Collezione& cc)
{
    size = cc.size;
    contents = cc.contents;
    if (size > 0)
        c = new InsiemeDiInteri[size];
    for (int i = 0; i < contents; i++)
        c[i] = cc.c[i];
}

Collezione::operator int() const
{
    int retval = 0;
    for (int i = 0; i < size; i++)
        retval += c[i];
    return retval;
}

Collezione& Collezione::operator=(const Collezione& cc)
{

```

```
    if (&cc != this) {
        if (size != cc.size) {
            if (size > 0)
                delete [] c;
            size = cc.size;
            if (size > 0)
                c = new InsiemeDiInteri[size];
        }
        contents = cc.contents;
        for (int i = 0; i < contents; i++)
            c[i] = cc.c[i];
    }
    return *this;
}

Collezione& Collezione::operator--(InsiemeDiInteri idi)
{
    for (int i = contents - 1; i >= 0; i--)
        if (c[i] == idi) {
            while (++i < contents)
                c[i - 1] = c[i];
            break;
        }
    return *this;
}

ostream& operator<<(ostream& os, const Collezione& cc)
{
    if (cc.contents > 0)
        os << cc.c[0];
    for (int i = 1; i < cc.contents; i++)
        os << ' ' << cc.c[i];
    return os;
}
```

Esercizio n. 9

Un `Elemento` ha un *peso* di tipo intero e uno *stato* `TRASPARENTE` od `OPACO`. Le operazioni che possono essere effettuate su un elemento sono le seguenti:

- `Elemento e()`
Costruttore di default, che inizializza un elemento `e`. Inizialmente, tale elemento è trasparente ed ha peso 0.
- `Elemento e(s, n)`
Costruttore che inizializza un elemento `e`. Inizialmente, tale elemento ha stato `s` e peso `n`.
- `Elemento e1(e)`
Costruttore di copia, che inizializza un elemento `e1` col valore dell'elemento `e`.
- `e1 = e`
Operatore di assegnamento, che sostituisce il valore dell'elemento risultato `e1` con quello dell'elemento `e`.
- `recente()`
Operazione che ritorna il valore che l'elemento creato più di recente possedeva al momento della sua creazione.
- `e.peso()`
Operazione che ritorna il peso dell'elemento `e`.
- `e.stato()`
Operazione che ritorna lo stato dell'elemento `e`.
- `cout << e`
Operatore di uscita per il tipo `Elemento`. L'uscita specifica lo stato, 'T' per trasparente ed 'O' per opaco, ed il peso dell'elemento, separati dal carattere ':'. Esempio:
T: 100
- `cin >> e`
Operatore di ingresso per il tipo `Elemento`. L'operatore legge il risultato dell'operatore di uscita per il tipo `Elemento`.
- `~Elemento()`
Distruttore.

Uno `StackDiElementi` è in grado di contenere elementi in numero limitato (la *capacità* dello stack di elementi). Le operazioni che possono essere effettuate su uno stack di elementi sono le seguenti:

- `StackDiElementi t()`
Costruttore di default, che inizializza uno stack di elementi `t` avente capacità pari al valore dell'intero costante `M`. Inizialmente, tale stack di elementi è vuoto.
- `StackDiElementi t(n)`
Costruttore che inizializza uno stack di elementi `t` avente capacità pari a `n` elementi. Inizialmente, tale stack di elementi è vuoto.
- `StackDiElementi t1(t)`
Costruttore di copia, che inizializza uno stack di elementi `t1` col valore dello stack di elementi `t`.
- `t1 = t`
Operatore di assegnamento, che sostituisce il valore dello stack di elementi risultato `t1` con quello dello stack di elementi `t`.

- `t.push()`
Operazione che inserisce nello stack di elementi `t` un elemento avente valore pari al valore che l'elemento creato più di recente possedeva al momento della sua creazione. L'operazione opera in accordo con la strategia LIFO.
- `t.push(e)`
Operazione che inserisce nello stack di elementi `t` un elemento avente valore pari al valore dell'elemento `e`. L'operazione opera in accordo con la strategia LIFO.
- `t.pop(s)`
Operazione che estrae dallo stack di elementi `t` il primo elemento di stato `s` e ritorna il valore di tale elemento. L'operazione opera in accordo con la strategia LIFO.
- `t.top(s)`
Operazione che ritorna il valore del primo elemento di stato `s` dello stack di elementi `t`. L'operazione opera in accordo con la strategia LIFO.
- `t.vuoto()`
Operazione che ritorna l'intero 1 se lo stack di elementi `t` è vuoto, e 0 altrimenti.
- `t.pieno()`
Operazione che ritorna l'intero 1 se lo stack di elementi `t` è pieno, e 0 altrimenti.
- `~StackDiElementi()`
Distruttore.

Mediante il linguaggio C++, realizzare il tipo di dati astratti `Elemento`, definito dalle precedenti specifiche. Utilizzare il tipo di dati astratti `Elemento` per realizzare il tipo di dati astratti `StackDiElementi` facendo ricorso ad una rappresentazione dello stack di elementi *ad array*. Individuare le eventuali situazioni di errore, e metterne in opera un corretto trattamento.

Soluzione (a cura di F. Potortì)

```
#include <iostream.h>
#include <stdlib.h>

void errore(const char *s);

class Elemento {
public:
    enum stato_t {invalid, T, O};
private:
    static stato_t ult_st;
    static int ult_p;
    stato_t st;
    int p;
public:
    Elemento(stato_t stato = T, int peso = 0) : st(stato), p(peso)
    {
        ult_st = st;
        ult_p = p;
    }
    Elemento(const Elemento& e) : st(e.st), p(e.p)
    {
        ult_st = st;
        ult_p = p;
    }
    static Elemento recente()
    {
        if (ult_st == invalid)
            errore("nessun elemento creato");
    }
};
```

```

        return Elemento(ult_st, ult_p);
    }
    int peso() const
    {
        return p;
    }
    stato_t stato() const
    {
        return st;
    }
};

class StackDiElementi {
    static const int M;
    int size;
    int Ttop;
    int Obottom;
    Elemento *stack;
    void copia(const StackDiElementi& s);
public:
    StackDiElementi(int n = M)
    {
        if (n > 0)
            stack = new Elemento[size = n];
        else
            size = 0;
        Ttop = 0;
        Obottom = size - 1;
    }
    StackDiElementi(const StackDiElementi& s)
    {
        if ((size = s.size) > 0)
            stack = new Elemento[size];
        copia(s);
    }
    StackDiElementi& operator=(const StackDiElementi&);
    void push(Elemento = Elemento::recente());
    Elemento pop(Elemento::stato_t);
    Elemento top(Elemento::stato_t) const;
    int vuoto() const
    {
        return (size == 0 || Obottom - Ttop + 1 == size);
    }
    int pieno () const
    {
        return (size == 0 || Ttop > Obottom);
    }
    ~StackDiElementi() {
        if (size > 0)
            delete [] stack;
    }
};

int Elemento::ult_p;
Elemento::stato_t Elemento::ult_st = Elemento::invalid;
const int StackDiElementi::M = 10;

void errore(const char *s)
{
    cerr << "StackDiElementi: " << s << endl;
    exit(1);
}

ostream& operator<<(ostream& os, const Elemento& e)
{
    os << ((e.stato() == Elemento::T) ? "T:" : "O:") << e.peso();
}

```

```

    return os;
}

istream& operator>>(istream& is, Elemento& e)
{
    Elemento::stato_t s;
    int peso;
    char c;
    if (!(is >> c))
        return is;
    switch (c) {
        case 'T':
            s = Elemento::T;
        case 'O':
            s = Elemento::O;
        default:
            is.clear(ios::failbit);
            return is;
    }
    if (!is.get(c))
        return is;
    if (c != ':') {
        is.clear(ios::failbit);
        return is;
    }
    if (is >> peso)
        e = Elemento(s, peso);
    return is;
}

void StackDiElementi::copia(const StackDiElementi& s)
{
    for (Ttop = 0; Ttop < s.Ttop; Ttop++)
        stack[Ttop] = s.stack[Ttop];
    for (Obottom = size - 1; Obottom > s.Obottom; Obottom--)
        stack[Obottom] = s.stack[Obottom];
}

StackDiElementi& StackDiElementi::operator=(const StackDiElementi& s)
{
    if (this != &s) {
        if (size != s.size) {
            if (size > 0)
                delete [] stack;
            if ((size = s.size) > 0)
                stack = new Elemento[size];
        }
        copia(s);
    }
    return *this;
}

void StackDiElementi::push(Elemento e)
{
    if (pieno())
        errore("push: stack pieno");
    if (e.stato() == Elemento::T)
        stack[Ttop++] = e;
    else
        stack[Obottom--] = e;
}

Elemento StackDiElementi::pop(Elemento::stato_t s)
{
    if (s == Elemento::T) {
        if (Ttop == 0)

```

```
        errore("pop: nessun elemento trasparente");
        return stack[--Ttop];
    }
    else {
        if (Obottom == size-1)
            errore("pop: nessun elemento opaco");
        return stack[++Obottom];
    }
}

Elemento StackDiElementi::top(Elemento::stato_t s) const
{
    if (s == Elemento::T) {
        if (Ttop == 0)
            errore("top: nessun elemento trasparente");
        return stack[Ttop - 1];
    }
    else {
        if (Obottom == size - 1)
            errore("top: nessun elemento opaco");
        return stack[Obottom + 1];
    }
}
```

ALTRI ESERCIZI

Esercizio n. 10

Utilizzando il linguaggio C++, realizzare il tipo di dati astratti `Percorso`, definito dalle specifiche dell'Esercizio n. 2. Ipotizzare che il numero di caselle di un percorso non superi il numero dei bit che formano un `long int`. Fare ricorso ad una rappresentazione del percorso a vettore di bit.

Esercizio n. 11

Un `DoppioStack` è formato da due stack di caratteri, che chiameremo stack di lato destro e stack di lato sinistro. Ciascuno stack ha capacità illimitata. Le operazioni che possono essere effettuate su un doppio stack sono le seguenti:

- `DoppioStack d()`
Costruttore di default, che inizializza un doppio stack `d`.
- `DoppioStack d1(d)`
Costruttore di copia, che inizializza un doppio stack `d1` col valore del doppio stack `d`.
- `d1 = d`
Operatore di assegnamento, che sostituisce il valore del doppio stack risultato `d1` con quello del doppio stack `d`.
- `d.push(t, c)`
Operazione che inserisce il carattere `c` nello stack di lato `t` del doppio stack `d`.
- `d.pop(t, n)`
Operazione che estrae `n` caratteri dallo stack di lato `t` del doppio stack `d`, e ritorna l'ultimo carattere estratto (gli altri caratteri estratti sono pertanto persi).
- `d.muovi(t, n, r)`
Operazione che estrae `n` caratteri dallo stack di lato `t` e li inserisce nell'altro stack del doppio stack `d`. Il parametro `r` specifica se i caratteri vengono inseriti nello stesso ordine (`r = 0`) o nell'ordine inverso (`r = 1`) a quello di estrazione.
- `d.caratteri(t)`
Operazione che ritorna il numero di caratteri contenuti nello stack di lato `t` del doppio stack `d`.
- `~DoppioStack()`
Distruttore.
- `cout << d`
Operatore di uscita per il tipo `DoppioStack`. L'uscita ha la forma seguente:

```
<"qwerty", "asd">
```

 In questo esempio, lo stack di lato sinistro del doppio stack `d` contiene in tutto sei caratteri, il carattere 'q' è stato inserito per primo ed il carattere 'y' è stato inserito per ultimo; lo stack di lato destro contiene in tutto tre caratteri, il carattere 'a' è stato inserito per primo ed il carattere 'd' è stato inserito per ultimo.
- `cin >> d`
Operatore di ingresso per il tipo `DoppioStack`. L'operatore legge il risultato dell'operatore di

uscita per il tipo `DoppioStack`.

Utilizzando il linguaggio C++, realizzare il tipo di dati astratti `DoppioStack`, definito dalle precedenti specifiche. Fare ricorso ad una rappresentazione del doppio stack a lista.

Esercizio n. 12

Un `Sistema` è formato da risorse, in numero limitato (la *capacità* del sistema). Ciascuna risorsa ha un nome, che consiste in una stringa di caratteri. Una risorsa può essere libera od occupata. Le operazioni che possono essere effettuate su un sistema sono le seguenti:

- `Sistema s(M)`
Costruttore che inizializza un sistema `s` avente capacità pari a `M` risorse. Inizialmente, il sistema non contiene risorse.
- `Sistema s1(s)`
Costruttore di copia, che inizializza un sistema `s1` col valore del sistema `s`.
- `s1 = s`
Operatore di assegnamento, che sostituisce il valore del sistema risultato `s1` con quello del sistema `s`. La capacità di `s1` è resa pari a quella di `s`.
- `s.aggiungi(c)`
Operazione che aggiunge una risorsa di nome `c` al sistema `s`. Tale risorsa è libera. L'operazione fallisce se il sistema già contiene un numero di risorse pari alla propria capacità. L'operazione fallisce inoltre se il sistema già contiene una risorsa di nome `c`. L'operazione ritorna 1 in caso di successo e 0 in caso di fallimento.
- `s.occupa(n)`
Operazione che occupa le `n` risorse del sistema `s` che sono libere da più tempo. A parità di tempo, l'operazione occupa le risorse in ordine alfabetico. L'operazione fallisce se nel sistema `s` non vi sono almeno `n` risorse libere, nel qual caso nessuna risorsa viene occupata. L'operazione ritorna 1 in caso di successo e 0 in caso di fallimento.
- `s.libera(n)`
Operazione che libera le `n` risorse del sistema `s` che sono occupate da più tempo. A parità di tempo, l'operazione libera le risorse in ordine alfabetico. L'operazione fallisce se nel sistema `s` non vi sono almeno `n` risorse occupate, nel qual caso nessuna risorsa viene liberata. L'operazione ritorna 1 in caso di successo e 0 in caso di fallimento.
- `s.contiene(c)`
Operazione che verifica se il sistema `s` contiene una risorsa di nome `c`. Se la verifica ha successo, l'operazione ritorna 1, altrimenti l'operazione ritorna 0.
- `s.seiLibera(c)`
Operazione che verifica se il sistema `s` contiene una risorsa *libera* di nome `c`. Se la verifica ha successo, l'operazione ritorna 1, altrimenti l'operazione ritorna 0.
- `s.elimina(c)`
Operazione che elimina la risorsa libera di nome `c` dal sistema `s`. L'operazione fallisce se il sistema non contiene una risorsa di nome `c`, o se tale risorsa è occupata. L'operazione ritorna 1 in caso di successo e 0 in caso di fallimento.

- `~Sistema()`
Distruttore.

Utilizzando il linguaggio C++, realizzare il tipo di dati astratti `Sistema`, definito dalle precedenti specifiche.

Esercizio n. 13

Un `Ambiente` è in grado di contenere un numero illimitato di elementi aventi valore intero maggiore di 0. Le operazioni che possono essere effettuate su un ambiente sono le seguenti:

- `Ambiente m()`
Costruttore di default, che inizializza un ambiente `m`. Tale ambiente è vuoto.
- `Ambiente m1(m)`
Costruttore di copia, che inizializza un ambiente `m1` col valore dell'ambiente `m`.
- `m1 = m`
Operatore di assegnamento, che sostituisce il valore dell'ambiente risultato `m1` con quello dell'ambiente `m`.
- `m.inserisci(i)`
Operazione che inserisce nell'ambiente `m` un nuovo elemento avente valore `i`. L'operazione fallisce se `m` già contiene un elemento avente valore `i`; in tal caso, l'inserzione non viene effettuata.
- `m.estrailTempo()`
Operazione che estrae dall'ambiente `m` l'elemento inserito da più tempo, e ritorna il valore di tale elemento. L'operazione fallisce se `m` è vuoto; in tal caso, l'operazione ritorna 0.
- `m.estrailValore()`
Operazione che estrae dall'ambiente `m` l'elemento avente il valore più basso, e ritorna tale valore. L'operazione fallisce se `m` è vuoto; in tal caso, l'operazione ritorna 0.
- `m.quantit()`
Operazione che ritorna il numero degli elementi contenuti nell'ambiente `m`.
- `~Ambiente()`
Distruttore.

Utilizzando il linguaggio C++, realizzare il tipo di dati astratti `Ambiente`, definito dalle precedenti specifiche. Associare al valore di ciascun elemento di un ambiente due puntatori, che chiameremo *puntatore di valore* e *puntatore di tempo*. I puntatori di valore degli elementi di uno stesso ambiente collegano tali elementi in una lista, la lista dei valori, nella quale gli elementi sono ordinati secondo i loro valori. I puntatori di tempo collegano gli elementi in una lista, la lista dei tempi, nella quale gli elementi sono ordinati secondo i loro tempi di inserzione.

Esercizio n. 14

Una `Riga` è formata da caselle numerate a partire da 1. Il numero delle caselle che formano una riga è detto la *dimensione* della riga. Tale dimensione deve essere maggiore od uguale ad 1. Ciascuna casella è colorata, ed i possibili colori sono BIANCO, ROSSO, GIALLO, VERDE e NERO. Le operazioni che possono essere effettuate su una riga sono le seguenti:

- Riga $r()$
Costruttore di default, che inizializza una riga r formata da una sola casella di colore BIANCO.
- Riga $r(n)$
Costruttore che inizializza una riga r formata da n caselle di colore BIANCO.
- Riga $r(n, c)$
Costruttore che inizializza una riga r formata da n caselle di colore c .
- Riga $r1(r)$
Costruttore di copia, che inizializza una riga $r1$ col valore della riga r .
- $r1 = r$
Operatore di assegnamento, che sostituisce il valore della riga risultato $r1$ con quello della riga r .
- $r1 += r2$
Operatore di somma e assegnamento, che aggiunge alla riga $r1$ un numero di caselle pari alla dimensione della riga $r2$. Le caselle aggiunte hanno i più alti numeri d'ordine e colore BIANCO.
- $r1 -= r2$
Operatore di sottrazione e assegnamento, che elimina dalla riga $r1$ un numero di caselle pari alla dimensione della riga $r2$. Le caselle eliminate sono quelle aventi i più bassi numeri d'ordine.
- $r1 \% = r2$
Operatore di modulo e assegnamento, che aggiunge caselle alla riga $r1$ o elimina caselle dalla riga $r1$ in modo tale che $r1$ abbia dimensione pari alla dimensione di $r2$. Se $r1$ ha già dimensione pari alla dimensione di $r2$, l'operatore lascia $r1$ inalterata. In caso di aggiunta di caselle, tali caselle aggiunte hanno i più alti numeri d'ordine e colore BIANCO. In caso di eliminazione di caselle, le caselle eliminate sono quelle aventi i più bassi numeri d'ordine.
- $r \% = n$
Operatore di modulo e assegnamento, che aggiunge caselle alla riga r o elimina caselle dalla riga r in modo tale che r abbia dimensione pari ad n . Se r ha già dimensione pari ad n , l'operatore lascia r inalterata. In caso di aggiunta di caselle, tali caselle aggiunte hanno i più alti numeri d'ordine e colore BIANCO. In caso di eliminazione di caselle, le caselle eliminate sono quelle aventi i più bassi numeri d'ordine.
- $r1 + r2$
Operatore di somma, che ritorna la riga ottenuta aggiungendo alla riga $r1$ un numero di caselle pari alla dimensione della riga $r2$. Le caselle aggiunte hanno i più alti numeri d'ordine e colore BIANCO.
- $r1 - r2$
Operatore di sottrazione, che ritorna la riga ottenuta eliminando dalla riga $r1$ un numero di caselle pari alla dimensione della riga $r2$. Le caselle eliminate sono quelle aventi i più bassi numeri d'ordine.
- $r1 \% r2$
Operatore modulo, che ritorna la riga ottenuta aggiungendo caselle alla riga $r1$ o eliminando caselle dalla riga $r1$ in modo da ottenere una riga avente dimensione pari alla dimensione di $r2$. Se $r1$ ha dimensione pari alla dimensione di $r2$, l'operatore ritorna il valore di $r1$. In caso di aggiunta di caselle, tali caselle aggiunte hanno i più alti numeri d'ordine e colore BIANCO. In caso di eliminazione di caselle, le caselle eliminate sono quelle aventi i più bassi numeri d'ordine.
- $r \% n$
Operatore modulo, che ritorna la riga ottenuta aggiungendo caselle alla riga r o eliminando caselle

dalla riga r in modo da ottenere una riga avente dimensione pari ad n . Se r ha dimensione pari ad n , l'operatore ritorna il valore di r . In caso di aggiunta di caselle, tali caselle aggiunte hanno i più alti numeri d'ordine e colore BIANCO. In caso di eliminazione di caselle, le caselle eliminate sono quelle aventi i più bassi numeri d'ordine.

- `r.colora(i, c)`
Operazione che assegna il colore c alla i -esima casella della riga r .
- `r.dimensione()`
Operazione che ritorna la dimensione della riga r .
- `~Riga()`
Distruttore.
- `cout << r`
Operatore di uscita per il tipo `Riga`. L'uscita ha la forma seguente:

```
[B, V, B, B, R, G, G, N]
```

I caratteri 'B', 'R', 'G', 'V' ed 'N' rappresentano una casella di colore BIANCO, ROSSO, GIALLO, VERDE e NERO, rispettivamente.

Utilizzando il linguaggio C++, realizzare il tipo di dati astratti `Riga`, definito dalle precedenti specifiche. Si faccia ricorso ad una rappresentazione della riga *ad array*. Si individuino le eventuali situazioni di errore, e se ne metta in opera un corretto trattamento.

Esercizio n. 15

Uno Scaffale contiene al più N scatole, numerate a partire da 1. Ciascuna scatola può contenere al più 5 sfere colorate, ed i possibili colori sono BIANCO, ROSSO, GIALLO, VERDE e NERO. Una scatola non può mai contenere due o più sfere dello stesso colore. Le operazioni che possono essere effettuate su uno scaffale sono le seguenti:

- `Scaffale s()`
Costruttore di default, che inizializza uno scaffale s . Inizialmente, lo scaffale non contiene scatole.
- `Scaffale s(r)`
Costruttore che inizializza uno scaffale s contenente r scatole vuote.
- `Scaffale s1(s)`
Costruttore di copia, che inizializza uno scaffale $s1$ col valore dello scaffale s .
- `s1 = s`
Operatore di assegnamento, che sostituisce il valore dello scaffale risultato $s1$ con quello dello scaffale s .
- `s.aggiungi()`
Operazione che aggiunge una scatola vuota allo scaffale s .
- `s.togli()`
Operazione che toglie dallo scaffale s la scatola contenente il minor numero di sfere. A parità di numero di sfere, l'operazione toglie la scatola avente più basso numero d'ordine. L'operazione ritorna il numero di sfere contenute nella scatola tolta.
- `s.togli(i)`
Operazione che toglie la i -esima scatola dallo scaffale s . L'operazione ritorna il numero di sfere contenute in tale scatola.

- `s += c`
Operatore di somma e assegnamento, che inserisce una sfera di colore `c` nella prima scatola dello scaffale `s`. Se tale scatola già contiene una sfera di colore `c`, l'operatore lascia lo scaffale inalterato.
- `s -= c`
Operatore di sottrazione e assegnamento, che estrae la sfera di colore `c` dalla prima scatola dello scaffale `s`. Se tale scatola non contiene una sfera di colore `c`, l'operatore lascia lo scaffale inalterato.
- `s.scambia(i, j)`
Operazione che scambia il contenuto della `i`-esima scatola con quello della `j`-esima scatola dello scaffale `s`.
- `s.contiene(i, c)`
Operazione che ritorna 1 se la `i`-esima scatola dello scaffale `s` contiene una sfera di colore `c`, e 0 altrimenti.
- `~Scaffale()`
Distruttore.
- `cout << s`
Operatore di uscita per il tipo `Scaffale`. L'uscita ha la forma seguente:

```
[RN], [G], [BRGVN]
```

Le parentesi quadre indicano una scatola. I caratteri 'B', 'R', 'G', 'V' ed 'N' rappresentano una sfera di colore BIANCO, ROSSO, GIALLO, VERDE e NERO, rispettivamente.

Utilizzando il linguaggio C++, realizzare il tipo di dati astratti `Scaffale`, definito dalle precedenti specifiche. Fare ricorso ad una rappresentazione che utilizzi un vettore di bit per ciascuna scatola. Si individuino le eventuali situazioni di errore, e se ne metta in opera un corretto trattamento.

Esercizio n. 16

Un `Paese` è formato da strade, ciascuna delle quali è dotata di lampioni per l'illuminazione. Ciascun lampiono può essere funzionante o guasto. Le operazioni che possono essere effettuate su un paese sono le seguenti:

- `Paese p()`
Costruttore di default, che inizializza un paese `p`. Inizialmente, il paese è privo di strade.
- `Paese p1(p)`
Costruttore di copia, che inizializza un paese `p1` col valore del paese `p`.
- `p1 = p`
Operatore di assegnamento, che sostituisce il valore dello paese risultato `p1` con quello del paese `p`.
- `p.nuovaStrada(s, n)`
Operazione che aggiunge una nuova strada di nome `s` al paese `p`. La strada è dotata di `n` lampioni, inizialmente tutti funzionanti.
- `p.lampioneGuasto(s, i)`
Operazione che registra il guasto dello `i`-esimo lampiono della strada `s` del paese `p`.

- `p.lampioneRiparato(s, i)`
Operazione che registra la riparazione dello i -esimo lampione della strada s del paese p .
- `~Paese()`
Distruttore.
- `cout << p`
Operatore di uscita per il tipo `Paese`. L'uscita ha la forma seguente:

```
Garibaldi, 24 [2, 3, 6]
Mazzini, 12
```

Le strade vengono elencate in ordine alfabetico. Le parentesi quadre contengono l'elenco dei lampioni guasti. In questo esempio, il paese ha due strade: via Garibaldi con 24 lampioni, dei quali sono guasti il secondo, il terzo ed il sesto; e via Mazzini con 12 lampioni tutti funzionanti.

Utilizzando il linguaggio C++, realizzare il tipo di dati astratti `Paese`, definito dalle precedenti specifiche. Si definisca in modo appropriato e si realizzi il tipo di dati astratti `Strada`, e si rappresenti un paese mediante una lista di strade. Si individuino le eventuali situazioni di errore, e se ne metta in opera un corretto trattamento.

Esercizio n. 17

Un `Palazzo` è formato da piani numerati a partire da 0 (il piano terra). Il piano terra presenta porta e finestre, gli altri piani solamente finestre. Le finestre di un dato piano sono numerate a partire da 1. Ciascuna finestra ha vetri e persiane, e le persiane possono venire aperte o chiuse solo se i vetri sono aperti. Le operazioni che possono essere effettuate su un palazzo sono le seguenti:

- `Palazzo p()`
Costruttore di default, che inizializza un palazzo p formato dal solo piano terra, che presenta la sola porta. Inizialmente, la porta è chiusa.
- `Palazzo p(V, M, f)`
Costruttore che inizializza un palazzo p di $V+1$ piani. Ogni piano escluso il piano terra presenta M finestre. Il piano terra presenta $M-1$ finestre e la porta, che è collocata in f -esima posizione (cioè dopo la $(f-1)$ -esima finestra). Inizialmente, la porta, i vetri e le persiane delle finestre sono chiusi.
- `Palazzo p1(s)`
Costruttore di copia, che inizializza un palazzo $p1$ col valore del palazzo p .
- `p1 = p`
Operatore di assegnamento, che sostituisce il valore dello palazzo risultato $p1$ con quello del palazzo p .
- `p.apriVetri(n, s)`
Operazione che apre i vetri della s -esima finestra del piano n del palazzo p .
- `p.chiudiVetri(n, s)`
Operazione che chiude i vetri della s -esima finestra del piano n del palazzo p .
- `p.apriPersiana(n, s)`
Operazione che apre le persiane della s -esima finestra del piano n del palazzo p .
- `p.chiudiPersiana(n, s)`
Operazione che chiude le persiane della s -esima finestra del piano n del palazzo p .

- `p.apriPorta()`
Operazione che apre la porta del palazzo `p`.
- `p.chiudiPorta()`
Operazione che chiude la porta del palazzo `p`.
- `p += n`
Operatore di somma e assegnamento, che apre i vetri di tutte le finestre del piano `n` del palazzo `p`.
- `p -= n`
Operatore di sottrazione e assegnamento, che chiude i vetri di tutte le finestre del piano `n` del palazzo `p`.
- `~Palazzo()`
Distruttore.
- `cout << p`
Operatore di uscita per il tipo `Palazzo`. L'uscita ha la forma seguente:

```
[Vp, VP, Vp, VP, Vp, VP]
[vP, vP, T, vP, vP, vP]
```

Le parentesi quadre indicano un piano. I caratteri 'v', 'V', 'p', 'P', 't' e 'T' rappresentano un vetro chiuso, un vetro aperto, una persiana chiusa, una persiana aperta, una porta chiusa ed una porta aperta, rispettivamente. In questo esempio, il palazzo presenta al piano terra la porta e 5 finestre, al primo piano 6 finestre. Al piano terra, la porta è collocata in terza posizione (dopo la seconda finestra) ed è aperta. Tutte le finestre del piano terra hanno vetri chiusi e persiane aperte. Al primo piano, tutte le finestre hanno vetri aperti, e le persiane sono alternativamente chiuse e aperte.

Utilizzando il linguaggio C++, realizzare il tipo di dati astratti `Palazzo`, definito dalle precedenti specifiche. Si individuino le eventuali situazioni di errore, e se ne metta in opera un corretto trattamento.

Esercizio n. 18

Una Cantina contiene scatole di bottiglie di vino. Tutte le bottiglie di una data scatola contengono vino della stessa annata. Le operazioni che possono essere effettuate su una cantina sono le seguenti:

- `Cantina c()`
Costruttore di default, che inizializza una cantina `c`. Inizialmente, la cantina è vuota.
- `Cantina c1(c)`
Costruttore di copia, che inizializza una cantina `c1` col valore della cantina `c`.
- `c1 = c`
Operatore di assegnamento, che sostituisce il valore della cantina risultato `c1` con quello della cantina `c`.
- `c.aggiungiScatola(a, n)`
Operazione che aggiunge alla cantina `c` una scatola contenente `n` bottiglie di vino di annata `a`.
- `c.togliScatola(a)`
Operazione che toglie dalla cantina `c` la scatola contenente il minor numero di bottiglie di vino di annata `a`, e ritorna tale numero di bottiglie.
- `c.togliScatola()`
Operazione che toglie dalla cantina `c` la scatola contenente il minor numero di bottiglie di vino, e

ritorna tale numero di bottiglie.

- `c.quantBottiglie(a)`
Operazione che ritorna il numero di bottiglie di vino di annata `a` contenute nella cantina `c`.
- `c.quantBottiglie()`
Operazione che ritorna il numero di bottiglie di vino contenute nella cantina `c`.
- `~Cantina()`
Distruttore.
- `cout << c`
Operatore di uscita per il tipo `Cantina`. L'uscita ha la forma seguente:

[1986:12, 1993:18, 1995:6]

In questo esempio, la cantina `c` contiene **complessivamente** 12 bottiglie di vino del 1986, 18 bottiglie di vino del 1993, e 6 bottiglie di vino del 1995.

Utilizzando il linguaggio C++, realizzare il tipo di dati astratti `Cantina`, definito dalle precedenti specifiche. Definire in modo appropriato e realizzare il tipo di dati astratti `Scatola`, e rappresentare una cantina mediante una lista di scatole. Individuare le eventuali situazioni di errore, e metterne in opera un corretto trattamento.

Esercizio n. 19

Una `Colonna` è formata da uno o più cubi, sovrapposti in ordine di dimensioni decrescenti. Le operazioni che possono essere effettuate su una colonna sono le seguenti:

- `Colonna c()`
Costruttore di default, che inizializza una colonna `c` formata da un solo cubo di dimensione 1.
- `Colonna c(d)`
Costruttore che inizializza una colonna `c` formata da un solo cubo di dimensione `d`.
- `Colonna c1(c)`
Costruttore di copia, che inizializza una colonna `c1` col valore della colonna `c`.
- `c1 = c`
Operatore di assegnamento, che sostituisce il valore della colonna risultato `c1` con quello della colonna `c`.
- `c1 + c2`
Operatore di somma, che ritorna una colonna formata dai cubi della colonna `c1` e dai cubi della colonna `c2`.
- `c1 - c2`
Operatore di sottrazione, che ritorna la colonna ottenuta eliminando dalla colonna `c1` tanti cubi quanti sono quelli che formano la colonna `c2`. I cubi eliminati sono quelli più piccoli.
- `c % n`
Operatore modulo, che ritorna la colonna ottenuta eliminando cubi dalla colonna `c` in modo tale da ridurla ad altezza pari a `n` cubi. I cubi eliminati sono quelli più piccoli. Se il numero dei cubi che formano `c` è minore o uguale ad `n`, l'operatore ritorna una colonna formata da tutti i cubi di `c`.
- `c1 += c2`
Operatore di somma e assegnamento, che aggiunge alla colonna `c1` i cubi della colonna `c2`.

- `c1 -= c2`
Operatore di sottrazione e assegnamento, che elimina dalla colonna `c1` tanti cubi quanti sono quelli che formano la colonna `c2`. I cubi eliminati sono quelli più piccoli.
- `c %= n`
Operatore di modulo e assegnamento, che elimina cubi dalla colonna `c` in modo tale da ridurla ad altezza pari a `n` cubi. I cubi eliminati sono quelli più piccoli. Se il numero dei cubi che formano la colonna `c` è minore o uguale ad `n`, l'operatore lascia `c` inalterata.
- `c1 == c2`
Operatore di uguaglianza, che ritorna 1 se le colonne `c1` e `c2` sono formate dagli stessi cubi, e 0 altrimenti.
- `~Colonna()`
Distruttore.
- `cout << c`
Operatore di uscita per il tipo `Colonna`. L'uscita ha la forma seguente:

```
[10]
[14]
[22]
[30]
```

In questo esempio, la colonna è formata da 4 cubi. Tali cubi hanno dimensione 30, 22, 14 e 10, rispettivamente.

Utilizzando il linguaggio C++, realizzare il tipo di dati astratti `Colonna`, definito dalle precedenti specifiche. Si faccia ricorso ad una rappresentazione della colonna *ad array*, e si rappresentino le dimensioni dei cubi mediante numeri interi. Si individuino le eventuali condizioni di errore, e se ne metta in opera un corretto trattamento.

Esercizio n. 20

Un `Bosco` è formato da alberi di età massima pari a 20 anni. Le operazioni che possono essere effettuate su un bosco sono le seguenti:

- `Bosco b()`
Costruttore di default, che inizializza un bosco `b`. Inizialmente, il bosco non contiene alberi.
- `Bosco b(t, n)`
Costruttore che inizializza un bosco `b` formato da `n` alberi di età pari a `t` anni.
- `Bosco b1(b)`
Costruttore di copia, che inizializza un bosco `b1` col valore del bosco `b`.
- `b1 = b`
Operatore di assegnamento, che sostituisce il valore del bosco risultato `b1` con quello del bosco `b`.
- `b.pianta(t)`
Operazione che aggiunge un albero di età pari a `t` anni al bosco `b`.
- `b.pianta(t, n)`
Operazione che aggiunge `n` alberi di età pari a `t` anni al bosco `b`.
- `b.abbatti(t)`
Operazione che toglie dal bosco `b` tutti gli alberi di età maggiore o uguale a `t` anni, e ritorna il numero di tali alberi.

- `b.quant(t)`
Operazione che ritorna il numero di alberi del bosco `b` aventi età maggiore od uguale a `t` anni.
- `b.piantati()`
Operazione che ritorna il numero totale di alberi piantati nel bosco `b`.
- `b.abbattuti()`
Operazione che ritorna il numero totale di alberi abbattuti nel bosco `b`.
- `piantatiTutti()`
Operazione che ritorna il numero di alberi complessivamente piantati in tutti i boschi.
- `abbattutiTutti()`
Operazione che ritorna il numero di alberi complessivamente abbattuti in tutti i boschi.
- `~Bosco()`
Distruttore.
- `cout << b`
Operatore di uscita per il tipo `Bosco`. L'uscita ha la forma seguente:
[120:5, 12:10, 14:18]
In questo esempio, il bosco `b` contiene 120 alberi di età pari a 5 anni, 12 alberi di età pari a 10 anni, e 14 alberi di età pari a 18 anni.

Utilizzando il linguaggio C++, realizzare il tipo di dati astratti `Bosco`, definito dalle precedenti specifiche. Si individuino le eventuali situazioni di errore, e se ne metta in opera un corretto trattamento.

Esercizio n. 21

Un `Registro` è in grado di contenere elementi a valore intero. Il numero di tali elementi è limitato, ed è chiamato la capacità del registro. Ciascun elemento può essere libero o bloccato. Le operazioni che possono essere effettuate su un registro sono le seguenti:

- `Registro d()`
Costruttore di default, che inizializza un registro `d` avente capacità pari a 10 elementi. Inizialmente, il registro è vuoto.
- `Registro d(N)`
Costruttore che inizializza un registro `d` avente capacità pari a `N` elementi. Inizialmente, il registro è vuoto.
- `Registro d1(d)`
Costruttore di copia, che inizializza un registro `d1` col valore del registro `d`.
- `d1 = d`
Operatore di assegnamento, che sostituisce il valore del registro `d1` con quello del registro `d`.
- `d += i`
Operatore di somma e assegnamento, che inserisce nel registro `d` un elemento bloccato avente valore `i`.
- `d *= r`
Operatore di prodotto e assegnamento, che libera gli `r` elementi bloccati inseriti da più tempo nel registro `d`. Se il registro contiene meno di `r` elementi bloccati, l'operazione libera tali elementi.

- `d -= r`
Operatore di sottrazione e assegnamento, che elimina dal registro `d` gli `r` elementi liberi inseriti da più tempo. Se il registro contiene meno di `r` elementi liberi, l'operazione elimina tali elementi.
- `~d`
Operatore di complemento, che ritorna il valore dell'elemento inserito da più tempo nel registro `d`.
- `d.pieno()`
Operazione che ritorna l'intero 1 se il registro `d` è pieno, e 0 altrimenti.
- `d.vuoto()`
Operazione che ritorna l'intero 1 se il registro `d` è vuoto, e 0 altrimenti.
- `~Registro()`
Distruttore.
- `cin >> d`
Operatore di ingresso per il tipo `Registro`. L'ingresso ha la forma di un intero. Tale intero rappresenta il valore di un elemento bloccato che viene inserito nel registro `d`.
- `cout << d`
Operatore di uscita per il tipo `Registro`. L'uscita ha la forma di un intero che rappresenta il valore dell'elemento inserito da più tempo nel registro `d`.

Utilizzando il linguaggio C++, realizzare il tipo di dati astratti `Registro`, definito dalle precedenti specifiche. Si individuino le eventuali situazioni di errore, e se ne metta in opera un corretto trattamento.

Esercizio n. 22

Un `Cammino` è formato da posizioni numerate a partire da 1. Ciascuna posizione può essere libera o prenotata. Un pedone si sposta lungo il cammino nelle due direzioni di marcia possibili. Le operazioni che possono essere effettuate su un cammino sono le seguenti:

- `Cammino c()`
Costruttore di default, che inizializza un cammino `c` formato da due posizioni libere. Il pedone è collocato sulla prima posizione.
- `Cammino c(n)`
Costruttore che inizializza un cammino `c` formato da `n` posizioni libere. Il pedone è collocato sulla prima posizione.
- `Cammino c1(c)`
Costruttore di copia, che inizializza un cammino `c1` col valore del cammino `c`.
- `c1 = c`
Operatore di assegnamento, che sostituisce il valore del cammino risultato `c1` con quello del cammino `c`.
- `c %= n`
Operatore di modulo e assegnamento, che aggiunge posizioni al cammino `c` o elimina posizioni dal cammino `c` in modo tale che `c` risulti formato da `n` posizioni. Tutte le posizioni diventano libere. Il pedone si sposta sulla prima posizione.
- `c += s`
Operatore di somma e assegnamento, che prenota la `s`-esima posizione del cammino `c`.

- `c.spostaPedone()`
Operazione che sposta il pedone nel cammino `c`, sulla posizione prenotata più vicina nelle due direzioni. Tale posizione diventa libera. Se non vi sono posizioni prenotate, l'operazione lascia il cammino inalterato.
- `~Cammino()`
Distruttore.
- `cin >> c`
Operatore di ingresso per il tipo `Cammino`. L'ingresso ha la forma di un intero che rappresenta il numero di posizioni del cammino `c`. L'operazione modifica `c` in modo tale che esso risulti formato da tale numero di posizioni. Tutte le posizioni sono libere. Il pedone si colloca sulla prima posizione.
- `cout << c`
Operatore di uscita per il tipo `Cammino`. L'uscita ha la forma seguente:

```
x-x---P--x--
```


Il carattere '-' rappresenta una posizione libera, il carattere 'x' rappresenta una posizione prenotata, il carattere 'P' rappresenta il pedone. In questo esempio, il cammino è formato da 12 posizioni, delle quali sono prenotate la prima, la terza e la decima. Il pedone è collocato sulla settima posizione.

Utilizzando il linguaggio C++, realizzare il tipo di dati astratti `Cammino`, definito dalle precedenti specifiche. Si individuino le eventuali situazioni di errore, e se ne metta in opera un corretto trattamento.

Esercizio n. 23

Al Banco di un supermercato lavorano uno o più commessi. I clienti formano un'unica coda. Le operazioni che possono essere effettuate su un banco sono le seguenti:

- `Banco d()`
Costruttore di default, che inizializza un banco `d` al quale lavora un solo commesso. Inizialmente, tale commesso riposa.
- `Banco d(C)`
Costruttore che inizializza un banco `d` al quale lavorano `C` commessi. Inizialmente, tutti i commessi riposano.
- `Banco d1(d)`
Costruttore di copia, che inizializza un banco `d1` col valore del banco `d`.
- `d1 = d`
Operatore di assegnamento, che sostituisce il valore del banco risultato `d1` con quello del banco `d`.
- `d.successivo(i)`
Operazione che coinvolge lo `i`-esimo commesso del banco `d`. Tale commesso inizia a servire il successivo cliente in coda. Se non vi sono clienti in coda, il commesso riposa.
- `d.cliente()`
Operazione che aggiunge un nuovo cliente al banco `d`. Se vi sono commessi liberi, il cliente viene immediatamente servito da uno di tali commessi, altrimenti il cliente passa in coda.

- `d += n`
Operatore di somma e assegnamento, che aggiunge `n` commessi al banco `d`. Se vi sono clienti in coda, i nuovi commessi iniziano immediatamente a servirli, altrimenti riposano.
- `d -= n`
Operatore di sottrazione e assegnamento, che toglie `n` commessi dal banco `d`. Tali commessi devono essere liberi.
- `~Banco()`
Distruttore.
- `cout << d`
Operatore di uscita per il tipo `Banco`. L'uscita ha la forma seguente:
[LRLLL]
In questo esempio, il banco ha cinque commessi, dei quali il secondo ed il terzo riposano, e gli altri servono altrettanti clienti.

Utilizzando il linguaggio C++, realizzare il tipo di dati astratti `Banco`, definito dalle precedenti specifiche. Si individuino le eventuali situazioni di errore, e se ne metta in opera un corretto trattamento.

Esercizio n. 24

Una *Girandola* ha un petalo trasparente ed i rimanenti petali colorati. I possibili colori sono BIANCO, ROSSO, GIALLO e VERDE. Le operazioni che possono essere effettuate su una girandola sono le seguenti:

- `Girandola d()`
Costruttore di default, che inizializza una girandola `d` formata da un solo petalo. Tale petalo è trasparente.
- `Girandola d(n)`
Costruttore che inizializza una girandola `d` formata da `n` petali. Il primo petalo è trasparente ed i rimanenti petali hanno colore BIANCO.
- `Girandola d1(d)`
Costruttore di copia, che inizializza una girandola `d1` col valore della girandola `d`.
- `d1 = d`
Operatore di assegnamento, che sostituisce il valore della girandola risultato `d1` con quello della girandola `d`.
- `d %= n`
Operatore di modulo e assegnamento, che sposta il petalo trasparente della girandola `d` in avanti di `n` posizioni.
- `d % n`
Operatore di modulo, che ritorna la girandola ottenuta spostando il petalo trasparente della girandola `d` in avanti di `n` posizioni.
- `d += c`
Operatore di somma e assegnamento, che aggiunge alla girandola `d` un petalo di colore `c` nella posizione successiva al petalo trasparente.
- `d + c`
Operatore di somma, che ritorna la girandola ottenuta aggiungendo alla girandola `d` un petalo di

colore c nella posizione successiva al petalo trasparente.

- $d -= c$
Operatore di sottrazione e assegnamento, che elimina dalla girandola d il petalo di colore c più vicino alla posizione del petalo trasparente.
- $d - c$
Operatore di sottrazione, che ritorna la girandola ottenuta eliminando dalla girandola d il petalo di colore c più vicino alla posizione del petalo trasparente.
- `~Girandola()`
Distruttore.
- `cout << d`
Operatore di uscita per il tipo `Girandola`. L'uscita ha la forma seguente:

`[-RRGRVV]`

Il carattere '-' rappresenta il petalo trasparente, che viene sempre indicato per primo. I caratteri 'B', 'R', 'G' e 'V' rappresentano petali di colore BIANCO, ROSSO, GIALLO e VERDE, rispettivamente. In questo esempio, la girandola ha tre petali rossi, uno giallo e due verdi.

Utilizzando il linguaggio C++, realizzare il tipo di dati astratti `Girandola`, definito dalle precedenti specifiche. Si individuino le eventuali situazioni di errore, e se ne metta in opera un corretto trattamento.

Esercizio n. 25

Un `Segnapunti` è in grado di registrare i risultati del lancio di due dadi. Le operazioni che possono essere effettuate su un `segnapunti` sono le seguenti:

- `Segnapunti d()`
Costruttore di default, che inizializza un `segnapunti d`. Il `segnapunti` è in grado di mantenere traccia di quale è il risultato registrato più di recente. Inizialmente, il `segnapunti` non contiene alcuna registrazione.
- `Segnapunti d(R)`
Costruttore di default, che inizializza un `segnapunti d`. Il `segnapunti` è in grado di mantenere traccia di quali sono gli R risultati registrati più di recente. Inizialmente, il `segnapunti` non contiene alcuna registrazione.
- `Segnapunti d1(d)`
Costruttore di copia, che inizializza un `segnapunti d1` col valore del `segnapunti d`.
- $d1 = d$
Operatore di assegnamento, che sostituisce il valore del `segnapunti d1` con quello del `segnapunti d`.
- $d += p$
Operatore di somma e assegnamento, che registra nel `segnapunti d` il risultato p di un nuovo lancio di dadi.
- $d -= p$
Operatore di sottrazione e assegnamento, che elimina dal `segnapunti d` le registrazioni di tutti i lanci di dadi aventi risultato p .
- $d \% = n$
Operatore di modulo e assegnamento, che elimina dal `segnapunti d` le registrazioni dei risultati de-

gli n lanci di dadi più recenti, con $n \leq R$. Se n vale 0, l'operatore elimina dal segnapunti d tutte le registrazioni in esso contenute.

- `~Segnapunti()`
Distruttore.
- `cin >> d`
Operatore di ingresso per il tipo `Segnapunti`. L'ingresso ha la forma di un intero che rappresenta il risultato di un nuovo lancio di dadi. Tale risultato viene registrato nel segnapunti d .
- `cout << d`
Operatore di uscita per il tipo `Segnapunti`. L'uscita ha la forma seguente:
[2->1, 4->3, 12->2]
In questo esempio, il risultato 2 è stato ottenuto una volta, il 4 è stato ottenuto tre volte, e il dodici due volte.

Utilizzando il linguaggio C++, realizzare il tipo di dati astratti `Segnapunti`, definito dalle precedenti specifiche. Fare ricorso ad una rappresentazione del segnapunti ad array che dia luogo ad una realizzazione efficiente dell'operatore di uscita (`<<`). Individuare le eventuali situazioni di errore, e mettere in opera un corretto trattamento.

Esercizio n. 26

Un `Pavimento` è formato da mattoni bianchi o colorati, ed i possibili colori sono ROSSO, GIALLO e VERDE. I mattoni sono disposti in righe. Le operazioni che possono essere effettuate su un pavimento sono le seguenti:

- `Pavimento d()`
Costruttore di default, che inizializza un pavimento d formato da un solo mattone bianco.
- `Pavimento d(N, M)`
Costruttore che inizializza un pavimento d formato da N righe di M mattoni ciascuna. Inizialmente, tutti i mattoni sono bianchi.
- `Pavimento d1(d)`
Costruttore di copia, che inizializza un pavimento $d1$ col valore del pavimento d .
- `d1 = d`
Operatore di assegnamento, che sostituisce il valore del pavimento $d1$ con quello del pavimento d .
- `d.colora(i, j, c)`
Operazione che colora il j -esimo mattone della i -esima riga del pavimento d . Il mattone assume il colore c . L'operazione fallisce se tale mattone non è bianco. L'operazione ritorna un'indicazione di successo (1) o fallimento (0).
- `d.colora(j, c)`
Operazione che colora il j -esimo mattone di ciascuna riga del pavimento d . Tali mattoni assumono il colore c . L'operazione fallisce se uno o più di tali mattoni non è bianco. L'operazione ritorna un'indicazione di successo (1) o fallimento (0).
- `d.coloraRiga(i, c)`
Operazione che colora tutti i mattoni della i -esima riga del pavimento d . Tali mattoni assumono il colore c . L'operazione fallisce se uno o più di tali mattoni non è bianco. L'operazione ritorna un'indicazione di successo (1) o fallimento (0).

- `d += j`
Operatore di somma e assegnamento, che rende bianco il j -esimo mattone di ciascuna riga del pavimento `d`.
- `d *= i`
Operatore di prodotto e assegnamento, che rende bianchi tutti i mattoni della i -esima riga del pavimento `d`.
- `d % c`
Operatore di modulo, che ritorna un intero pari al numero dei mattoni di colore `c` del pavimento `d`.
- `~Pavimento()`
Distruttore.
- `cout << d`
Operatore di uscita per il tipo `Pavimento`. L'uscita ha la forma seguente:

```

- - G R
V V V V
- - V -

```

Il carattere '-' rappresenta un mattone bianco, i caratteri 'R', 'G' e 'V' rappresentano mattoni di colore ROSSO, GIALLO e VERDE, rispettivamente. In questo esempio, il pavimento è formato da 3 righe di 4 mattoni ciascuna, e tutti i mattoni della seconda riga sono di colore VERDE.

Utilizzando il linguaggio C++, realizzare il tipo di dati astratti `Pavimento`, definito dalle precedenti specifiche. Si individuino le eventuali situazioni di errore, e se ne metta in opera un corretto trattamento.

Esercizio n. 27

Un `Triangolo` è formato da caselle organizzate in linee numerate a partire da 1. La i -esima linea contiene i caselle. Ad esempio, in un triangolo a tre linee, la prima linea contiene una casella, la seconda linea contiene due caselle, e la terza tre caselle. Ciascuna casella è colorata, ed i possibili colori sono BIANCO e NERO. Le operazioni che possono essere effettuate su un triangolo sono le seguenti:

- `Triangolo t()`
Costruttore di default, che inizializza un triangolo `t`. Inizialmente, il triangolo è formato da una sola linea. La casella di tale linea è di colore BIANCO.
- `Triangolo t(r)`
Costruttore che inizializza un triangolo `t`. Inizialmente, il triangolo è formato da r linee. Le caselle di tali linee sono di colore BIANCO.
- `Triangolo t1(t)`
Costruttore di copia, che inizializza un triangolo `t1` col valore del triangolo `t`.
- `t1 = t`
Operatore di assegnamento, che sostituisce il valore del triangolo risultato `t1` con quello del triangolo `t`.
- `t += r`
Operatore di somma e assegnamento, che aggiunge r linee al triangolo `t`. Le caselle di tali linee hanno colore BIANCO.
- `t + r`
Operatore di somma, che ritorna il triangolo ottenuto aggiungendo r linee al triangolo `t`. Le caselle

di tali linee hanno colore BIANCO.

- `t -= r`
Operatore di sottrazione e assegnamento, che toglie `r` linee dal triangolo `t`.
- `t - r`
Operatore di sottrazione, che ritorna il triangolo ottenuto togliendo `r` linee dal triangolo `t`.
- `t.colora(i, j, c)`
Operazione che assegna il colore `c` alla `j`-esima casella della `i`-esima linea del triangolo `t`.
- `t.seiBianca(i, j)`
Operazione che ritorna 1 se la `j`-esima casella della `i`-esima linea del triangolo `t` è di colore BIANCO, e ritorna 0 altrimenti.
- `~Triangolo()`
Distruttore.
- `cout << t`
Operatore di uscita per il tipo `Triangolo`. L'uscita ha la forma seguente:

```
B
NB
BNN
BNNB
```

Il carattere 'B' rappresenta una casella di colore BIANCO, ed il carattere 'N' rappresenta una casella di colore NERO.

Utilizzando il linguaggio C++, realizzare il tipo di dati astratti `Triangolo`, definito dalle precedenti specifiche. Si individuino le eventuali situazioni di errore, e se ne metta in opera un corretto trattamento.

Esercizio n. 28

Un `Deposito` di liquidi è formato da recipienti di capacità diverse. La capacità si misura in litri. Nel deposito può esserci al più un recipiente parzialmente pieno (tutti gli altri recipienti sono completamente pieni oppure completamente vuoti). Le operazioni che possono essere effettuate su un deposito sono le seguenti:

- `Deposito d()`
Costruttore di default, che inizializza un deposito `d`. Inizialmente, il deposito non contiene recipienti.
- `Deposito d1(d)`
Costruttore di copia, che inizializza un deposito `d1` col valore del deposito `d`.
- `d1 = d`
Operatore di assegnamento, che sostituisce il valore del deposito risultato `d1` con quello del deposito `d`.
- `d.inserisciRecipiente(c)`
Operazione che inserisce un recipiente completamente vuoto avente capacità `c` nel deposito `d`.
- `d.eliminaRecipiente()`
Operazione che elimina dal deposito `d` il recipiente completamente vuoto inserito nel deposito da più tempo. Se il deposito `d` non contiene recipienti completamente vuoti, l'operazione lascia `d` inalterato.

- `d += q`
Operatore di somma e assegnamento, che aggiunge `q` litri di liquido al deposito `d`.
 - `d -= q`
Operatore di sottrazione e assegnamento, che estrae `q` litri di liquido dal deposito `d`.
 - `~Deposito()`
Distruttore.
 - `cout << d`
Operatore di uscita per il tipo `Deposito`. L'uscita ha la forma seguente:

```
<300:300> <300:300> <200:110> <400:0>
```

 Ogni coppia di parentesi angolate indica un recipiente. I recipienti vengono stampati secondo l'ordine di inserzione nel deposito. In questo esempio, il deposito `d` è formato da due recipienti di capacità pari a 300 litri completamente pieni, da un recipiente di capacità pari a 200 litri che contiene 110 litri di liquido, e da un recipiente di capacità pari a 400 litri completamente vuoto. Il recipiente di capacità pari a 400 litri è stato inserito per ultimo.
- Utilizzando il linguaggio C++, realizzare il tipo di dati astratti `Deposito`, definito dalle precedenti specifiche. Individuare le eventuali situazioni di errore, e metterne in opera un corretto trattamento.

Esercizio n. 29

Ciascuna colonna di un `Colonnato` è formata da al più `D` dischi sovrapposti. L'altezza di una colonna è data dal numero di dischi che la formano. Le colonne di un `colonnato` sono mantenute ordinate secondo altezze crescenti. I dischi sono colorati, ed i possibili colori sono `BIANCO` e `NERO`. Le operazioni che possono essere effettuate su un `colonnato` sono le seguenti:

- `Colonnato d()`
Costruttore di default, che inizializza un `colonnato d` che consiste in una sola colonna. Inizialmente, tale colonna è formata da un solo disco di colore `BIANCO`.
- `Colonnato d1(d)`
Costruttore di copia, che inizializza un `colonnato d1` col valore del `colonnato d`.
- `d1 = d`
Operatore di assegnamento, che sostituisce il valore del `colonnato risultato d1` con quello del `colonnato d`.
- `d += n`
Operatore di somma e assegnamento, che aggiunge `n` colonne al `colonnato d`. Ciascuna di tali colonne è formata da un solo disco di colore `BIANCO`.
- `d -= n`
Operatore di sottrazione e assegnamento, che elimina dal `colonnato d` le `n` colonne più basse.
- `d.aggiungiDisco(i, c)`
Operazione che aggiunge un disco di colore `c` nella posizione più alta della `i`-esima colonna del `colonnato d`.
- `d.eliminaDisco(i)`
Operazione che elimina il disco nella posizione più alta della `i`-esima colonna del `colonnato d`, e ritorna il colore di tale disco.

- `d.colore(i)`
Operazione che ritorna il colore del disco nella posizione più alta della i -esima colonna del colonnato `d`.
- `~Colonnato()`
Distruttore.

Utilizzando il linguaggio C++, definire in modo appropriato e realizzare il tipo di dati astratti `Colonna`, rappresentando una colonna mediante un `unsigned` interpretato come configurazione di bit (vettore di bit). Mediante il tipo di dati astratti `Colonna`, realizzare il tipo di dati astratti `Colonnato` definito dalle precedenti specifiche. Individuare le eventuali situazioni di errore, e metterne in opera un corretto trattamento.

Esercizio n. 30

Ciascuna parola di una lista di parole è formata al più T lettere dell'alfabeto. Le operazioni che possono essere effettuate su una lista di parole sono le seguenti:

- `ListaDiParole p()`
Costruttore di default, che inizializza una lista di parole `p`. Inizialmente, tale lista di parole è vuota.
- `ListaDiParole p1(p)`
Costruttore di copia, che inizializza una lista di parole `p1` col valore della lista di parole `p`.
- `p1 = p`
Operatore di assegnamento, che sostituisce il valore della lista di parole risultato `p1` con quello della lista di parole `p`.
- `p += r`
Operatore di somma e assegnamento, che inserisce la parola `r` nella lista di parole `p`.
- `p -= r`
Operatore di sottrazione e assegnamento, che elimina la parola `r` dalla lista di parole `p`. Se `p` non contiene la parola `r`, l'operatore lascia `p` inalterata.
- `p % c`
Operatore di modulo, che ritorna il numero di parole contenute nella lista di parole `p` che iniziano con la lettera `c`.
- `~ListaDiParole()`
Distruttore.
- `cin >> p`
Operatore di ingresso per il tipo `ListaDiParole`. L'ingresso consiste in parole separate da spaziature. Tali parole sono inserite nella lista `p`.
- `cout << p`
Operatore di uscita per il tipo `ListaDiParole`. L'uscita consiste nelle parole contenute nella lista di parole `p`, una parola per riga, in ordine alfabetico.

Utilizzando il linguaggio C++, realizzare il tipo di dati astratti `ListaDiParole`, definito dalle precedenti specifiche. Rappresentare la lista di parole mediante un vettore di puntatori: lo i -esimo puntatore riferirà la prima parola (in ordine alfabetico) che inizia con la i -esima lettera dell'alfabeto (ad esempio, il terzo puntatore riferirà la prima parola che inizia con la lettera 'c'). Individuare le eventuali situazioni di errore, e metterne in opera un corretto trattamento.