Lanfranco Lopriore, Franco Maria Nardini, Nicola Tonellotto

C++: Esercizi di Programmazione sulle Classi

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.

Indice

Esercizio n. 1 – Ruota

Una *Ruota* è divisa in spicchi numerati a partire da 1. Ciascuno spicchio 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.
- `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[1]

```
#include <iostream>
#include <stdlib.h>  // per exit()
using namespace std;

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

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 contrai(int);
   void espandi(int);
};

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)
{
   for (int i = 0; i < n; i++) {
      Spicchio *p = ultimo->prossimo;
```

[1] La soluzione di questo esercizio e di alcuni esercizi successivi comprende una funzione *main()*, che ha l'unico scopo di permettere l'esecuzione delle principali funzionalità dei tipi di dati astratti definiti negli esercizi stessi.

```
      ultimo->prossimo = p->prossimo;
      delete p;
   }
   quanti -= n;
}

void Ruota::espandi(int n)
{
   Spicchio *primo = ultimo->prossimo;
   for (int i = 0; i < n; i++) {
      ultimo->prossimo = new Spicchio;
      ultimo = ultimo->prossimo;
   }
   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 (int i = 0; i < quanti - 1; i++) {
         q = q->prossimo;
         p->prossimo = new Spicchio(q->stato);
```

```
            p = p->prossimo;
         }
         p->prossimo = ultimo;
      }
      return *this;
   }

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

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

   inline 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;
      if (q != ultimo)
         q = q->prossimo;
   }
   ultimo = 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;
}

int 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 (r2)\n"
"\tt (T) - per rendere trasparente uno spicchio di r1 (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 (r2)\n"
"\t{ (}) - per conoscere lo stato di uno spicchio di r1 (r2)\n"
"\t!     - 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': {
     r1.sovrapponi(r2);
     break;
   }
case '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 – Percorso

Un *Percorso* è formato da caselle numerate a partire da 1. Ciascuna casella può assumere il colore bianco o 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 una pedina arriva sull'ultima casella, esce dal percorso. Le operazioni che possono essere effettuate su un percorso sono le seguenti:

- `Percorso p;`
  Costruttore di default, che inizializza un percorso *p* di lunghezza pari a 10 caselle. Inizialmente, tutte le caselle sono di colore bianco.
- `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:

  ```
  bbBBBnBBNNb
  ```

  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

```
#include <iostream>
#include <stdlib.h>  // per exit()
using namespace std;
```

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

enum colore { BIANCO, NERO };

const int lunghezzaDiDefault = 10;

class Percorso {
   friend ostream& operator<<(ostream&, const Percorso&);

   class casella {
      enum stato { LIBERA, OCCUPATA } st;
      colore col;
   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();
};

int Percorso::avanza(int i)
// Sposta la pedina dalla casella i-esima alla casella più avanzata
// possibile
{
   int j;
   for (j = i + 1; j < lunghezza && !cc[j].seiOccupata() &&
      cc[i].qualeColore() == cc[j].qualeColore(); j++);
   return j - 1;
}

Percorso::Percorso(int lungh, colore c)
{
   if (lungh < 1)
      errore("Lunghezza errata");
   lunghezza = lungh;
   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)
// se la nuova pedina arriva sull'ultima casella, esce dal percorso
         cc[i].occupa();
   }
   return *this;
}

Percorso& Percorso::cambiaColore(int i)
{
   if (i < 1 || i > lunghezza)
      errore("Numero di casella errato");
   --i; // necessario perché le caselle sono numerate a partire da 1
   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;
}

inline 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;
}

int 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 (p2)\n"
"\tb (B) - per immettere una pedina bianca in p1 (p2)\n"
"\tc (C) - per cambiare il colore di una casella di p1 (p2)\n"
"\tp     - 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 – Contenitore

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 minuscola dell'alfabeto. Più elementi in 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 unitaria. Inizialmente il contenitore è vuoto.
- `Contenitore c(N);`
  Costruttore che inizializza un contenitore *c* avente capienza pari ad *N* elementi. Inizialmente il 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 somma e assegnamento, che inserisce nel contenitore *c* un elemento avente nome *p*.
- `c -= p`
  Operatore di sottrazione e assegnamento, che estrae dal contenitore *c* tutti gli elementi aventi nome *p*.
- `c % N`
  Operatore di 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 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 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.quanti()`
  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

```
#include <iostream>
#include <stdlib.h>  // per exit()
using namespace std;

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

class Contenitore {
   friend Contenitore
   operator*(const Contenitore&,const Contenitore&);
   friend Contenitore
   operator/(const Contenitore&,const Contenitore&);
   friend ostream& operator<<(ostream&, Contenitore);
   int numElem, cap;
   int ee[26];
public:
   Contenitore(int = 1);
// Contenitore(const Contenitore&);
// Contenitore& operator=(const Contenitore&);
   Contenitore& operator+=(char);
   Contenitore& operator-=(char);
   Contenitore operator%(int) const;
   int capienza() const { return cap; }
   int quanti() const { return numElem; }
// ~Contenitore();
};

Contenitore::Contenitore(int N)
{
   if (N <= 0)
      errore("Capienza errata");
   for (int i = 0; i <= 25; i++)
      ee[i] = 0;
   numElem = 0;
   cap = N;
}

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

Contenitore& Contenitore::operator-=(char p)
{
   if (p < 'a' || p > 'z')
      errore("Nome di elemento errato");
   numElem -= ee[p - 'a'];
   ee[p - 'a'] = 0;
   return *this;
}
```

```
Contenitore Contenitore::operator%(int N) const
{
   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++;
      }
   return c;
}

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

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

int 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 ottenere 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 << "Immetterne 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.quanti() << '\n';
              cout << "Capienza di c2: " << c2.capienza() << '\t'
                 << "Elementi in c2: " << c2.quanti() << '\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 – Rotore

Un *rotore* ha la forma di un cerchio diviso in settori. I settori sono numerati a partire da 0. Ciascun settore è in grado di contenere un numero illimitato di richieste. Ad ogni dato istante, un selettore è posizionato su uno dei settori del rotore. Le operazioni che possono essere effettuate su un rotore sono le seguenti:

- `Rotore r;`
  Costruttore di default, che inizializza un rotore *r* formato dal solo settore 0. Inizialmente, il settore 0 contiene una richiesta, e il selettore è posizionato su tale settore.
- `Rotore r(n);`
  Costruttore che inizializza un rotore *r* diviso in *n* settori. Inizialmente, il settore 0 contiene una richiesta, gli altri settori non contengono richieste, e il selettore è posizionato sul settore 0.
- `Rotore r1(r);`
  Costruttore di copia, che inizializza un rotore *r1* col valore del rotore *r*.
- `r1 = r`
  Operatore di assegnamento, che sostituisce il valore del rotore risultato *r1* con quello del rotore *r*.
- `r.inserisci(t)`
  Operazione che inserisce una nuova richiesta nel settore *t* del rotore *r*.
- `r.estrai()`
  Operazione che estrae una richiesta dal settore del rotore *r* sul quale è posizionato il selettore, e ritorna il numero di tale settore. L'operazione produce il movimento del selettore sul settore non privo di richieste più vicino, in senso orario o antiorario.
- `r % t`
  Operatore di modulo, che ritorna il numero di richieste presenti nel settore *t* del rotore *r*.
- `!r`
  Operatore di negazione logica, che ritorna il numero totale delle richieste presenti nel rotore *r*.
- `cout << r`
  Operatore di uscita per il tipo *Rotore*. L'uscita ha la forma seguente:

  ```
  4, [3], 0, 3, 6
  ```

  In questo esempio, il rotore ha cinque settori. Nel settore 0 sono presenti quattro richieste, nel settore 2 non sono presenti richieste. Il selettore è posizionato sul settore 1, nel quale sono presenti tre richieste.
- `~Rotore()`
  Distruttore.

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

Soluzione

```
#include <iostream>
using namespace std;

class Rotore {
   friend ostream& operator<<(ostream&, const Rotore&);
   int *settore;
   int dim, selettore;
public:
```

```
   Rotore(int = 1);
   Rotore(const Rotore&);
   Rotore& operator=(const Rotore&);
   Rotore& inserisci(int);
   int estrai();
   int operator%(int) const;
   int operator!() const;
   ~Rotore() { delete[] settore; }
};

Rotore::Rotore(int n)
{
   dim = (n > 0) ? n : 1;
   selettore = 0;
   settore = new int[dim];
   settore[0] = 1;
   for (int i = 1; i < dim; i++)
      settore[i] = 0;
}

Rotore::Rotore(const Rotore& r)
{
   dim = r.dim;
   selettore = r.selettore;
   settore = new int[dim];
   for (int i = 0; i < dim; i++)
      settore[i] = r.settore[i];
}

Rotore& Rotore::operator=(const Rotore& r)
{
   if (this != &r) {
      if (dim != r.dim) {
         delete[] settore;
         dim = r.dim;
         settore = new int[dim];
      }
      selettore = r.selettore;
      for (int i = 0; i < dim; i++)
         settore[i] = r.settore[i];
   }
   return *this;
}

Rotore& Rotore::inserisci(int t)
{
   if (t >= 0 && t < dim)
      settore[t]++;
   return *this;
}

int Rotore::estrai()
{
   int ris = -1;
   if (settore[selettore] > 0) {
      settore[selettore]--;
      ris = selettore;
   }
   for (int i = 1; i <= dim / 2; i++) {
      if (settore[(selettore + i) % dim] > 0) {
         selettore = (selettore + i) % dim;
         break;
```

```
      }
      if (settore[(dim + selettore - i) % dim] > 0) {
         selettore = (dim + selettore - i) % dim;
         break;
      }
   }
   return ris;
}

int Rotore::operator%(int t) const
{
   if (t >= 0 && t < dim)
      return settore[t];
   return -1;
}

int Rotore::operator!() const
{
   int conta = 0;
   for (int i = 0; i < dim; i++)
      conta += settore[i];
   return conta;
}

ostream& operator<<(ostream& os, const Rotore& r)
{
   for (int i = 0; i < r.dim; i++) {
      if (i == r.selettore)
         os << '[' << r.settore[i] << ']';
      else
         os << r.settore[i];
      if (i != r.dim - 1)
         os << ", ";
   }
   return os;
}
```

Esercizio n. 5 – Batteria

Una *pila* può contenere un numero limitato di valori interi positivi (maggiori o uguali a 0) gestiti secondo il criterio *last-in*, *first-out*. Il numero massimo di valori che possono essere contenuti in una pila è chiamato *capacità* della pila. A ciascuna pila sono associati un limite inferiore ed un limite superiore. Un valore intero è *compatibile* con una data pila se esso è compreso tra i limiti inferiore e superiore di quella pila. In un pila possono essere inseriti solo valori compatibili. Una batteria è formata da un insieme di pile numerate a partire da 0. Le operazioni che possono essere effettuate su una batteria sono le seguenti:

- `Batteria s;`
  Costruttore di default, che inizializza una batteria *s* formata da una sola pila avente capacità unitaria e limiti {0, 1}. Inizialmente, tale pila è vuota.
- `Batteria s(n, c, m);`
  Costruttore che inizializza una batteria *s* formata da *n* pile aventi capacità *c* e limiti {0, *m* - 1}, {*m*, 2*m* – 1}, {2*m*, 3*m* – 1},…. Inizialmente, tutte le pile della batteria *s* sono vuote.
- `Batteria s1(s);`
  Costruttore di copia, che inizializza una batteria *s1* col valore della batteria *s*.
- `s1 = s`
  Operatore di assegnamento, che sostituisce il valore della batteria risultato *s1* con quello della batteria *s*.
- `s.limiti(i, inf, sup)`
  Operazione che vuota la pila *i* della batteria *s* e ne modifica i limiti inferiore e superiore. Gli argomenti *inf* e *sup* rappresentano i nuovi limiti.
- `s.push(v)`
  Operazione che inserisce il valore intero positivo *v* nella batteria *s*. Il nuovo valore viene inserito nella pila non piena avente il numero d'ordine più basso tra le pile compatibili con *v*.
- `s.pop(v)`
  Operazione che estrae il valore intero positivo inserito più di recente nella pila non vuota avente il numero d'ordine più alto tra le pile della batteria *s* compatibili col valore intero positivo *v*. L'operazione ritorna il valore positivo estratto.
- `s.top(v)`
  Operazione che ritorna il valore intero positivo inserito più di recente nella pila non vuota avente il numero d'ordine più alto tra le pile della batteria *s* compatibili col valore intero positivo *v*.
- `s.pn(v)`
  Operazione che ritorna l'intero 1 se tutte le pile della batteria *s* compatibili con l'intero positivo *v* sono piene, e ritorna 0 altrimenti.
- `s.vt(v)`
  Operazione che ritorna l'intero 1 se tutte le pile della batteria *s* compatibili con l'intero positivo *v* sono vuote, e ritorna 0 altrimenti.
- `~Batteria()`
  Distruttore.

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

Soluzione

```
#include <iostream>
```

```
using namespace std;

class Batteria {

   class Pila {
      int cap, cima, inf, sup;
      int* elem;
      void copia(const Pila&);
   public:
      Pila(int = 1, int = 0, int = 1);
      Pila(const Pila& p) { copia(p); }
      Pila& operator=(const Pila&);
      int pop(int);
      int push(int);
      int top(int) const;
      int compatibile(int v) const { return (v >= inf && v <= sup); }
      void limiti(int i, int s) { inf = i; sup = s; cima = 0; }
      int piena() const { return (cima == cap); }
      int vuota() const { return (cima == 0); }
      ~Pila() { delete[] elem; }
   };

   int dim;
   Pila* pp;
   void copia(const Batteria&);
public:
   Batteria(int = 1, int = 1, int = 2);
   Batteria(const Batteria& b) { copia(b); }
   Batteria& operator=(const Batteria&);
   Batteria& limiti(int, int, int);
   Batteria& push(int);
   int pop(int);
   int top(int) const;
   int pn(int) const;
   int vt(int) const;
   ~Batteria() { delete[] pp; }
};

// ---------------- Classe Pila ---------------------- //

Batteria::Pila::Pila(int c, int i, int s)
{
   cap = c;
   inf = i;
   sup = s;
   cima = 0;
   elem = new int[cap];
}

void Batteria::Pila::copia(const Pila& p)
{
   cap = p.cap;
   inf = p.inf;
   sup = p.sup;
   cima = p.cima;
   elem = new int[cap];
   for (int i = 0; i < cima; i++)
      elem[i] = p.elem[i];
}

Batteria::Pila& Batteria::Pila::operator=(const Pila& p)
{
```

```
   if (this != &p) {
      delete[] elem;
      copia(p);
   }
   return *this;
}

int Batteria::Pila::pop(int v)
{
   if ( compatibile(v) && !vuota() )
      return elem[--cima];
   return -1;
}

int Batteria::Pila::push(int v)
{
   if ( compatibile(v) && !piena() ) {
      elem[cima++] = v;
      return 0;
   }
   return -1;
}

int Batteria::Pila::top(int v) const
{
   if ( compatibile(v) && !vuota() )
      return elem[cima - 1];
   return -1;
}

// ---------------- Classe Batteria ------------------ //

Batteria::Batteria(int n, int c, int m)
{
   dim = (n > 0) ? n : 1;
   c = (c > 0) ? c : 1;
   m = (m > 2) ? m : 2;
   pp = new Pila[dim];
   for (int i = 0; i < dim; i++)
      pp[i] = Pila(c, i * m, (i + 1) * m - 1);
}

void Batteria::copia(const Batteria& s)
{
   dim = s.dim;
   pp = new Pila[dim];
   for (int i = 0; i < dim; i++)
      pp[i] = s.pp[i];
}

Batteria& Batteria::operator=(const Batteria& s)
{
   if (this != &s) {
      delete[] pp;
      copia(s);
   }
   return *this;
}

Batteria& Batteria::limiti(int i, int inf, int sup)
{
   if (i >= 0 && i < dim && inf >= 0 && sup > inf)
```

```
      pp[i].limiti(inf, sup);
   return *this;
}

Batteria& Batteria::push(int v)
{
   if (v < 0)
      return -1;
   int i = 0;
   while (i < dim && pp[i].push(v) == -1)
      i++;
   return *this;
}

int Batteria::pop(int v)
{
   if (v < 0)
      return -1;
   int i = dim - 1;
   int ris = pp[i].pop(v);
   while (ris == -1 && i > 0)
      ris = pp[--i].pop(v);
   return ris;
}

int Batteria::top(int v) const
{
   if (v < 0)
      return -1;
   int i = dim - 1;
   int ris = pp[i].top(v);
   while (ris == -1 && i > 0)
      ris = pp[--i].top(v);
   return ris;
}

int Batteria::pn(int v) const
{
   for (int i = 0; i < dim; i++)
      if ( pp[i].compatibile(v) && !(pp[i].piena()) )
         return 0;
   return 1;
}

int Batteria::vt(int v) const
{
   for (int i = 0; i < dim; i++)
      if ( pp[i].compatibile(v) && !(pp[i].vuota()) )
         return 0;
   return 1;
}
```

Esercizio n. 6 – Terna multipla

Una terna è composta da tre campi aventi valore intero strettamente positivo. In alcune operazioni del tipo *Terna*, un *unsigned int*, chiamato *selettore*, viene interpretato come vettore di bit. I bit del selettore sono numerati a partire dal meno significativo (il bit 0). Il bit 0, se ad 1, specifica che l'operazione coinvolge il primo campo della terna, e similmente per il bit 1 e il secondo campo e per il bit 2 e il terzo campo della terna. Le operazioni che possono essere effettuate su una terna sono le seguenti:

- `Terna t;`
  Costruttore di default, che inizializza una terna *t* i cui campi hanno tutti valore unitario.
- `Terna t1(s);`
  Costruttore di copia, che inizializza la terna *t1* col valore della terna *t*.
- `t1 = t`
  Operatore di assegnamento, che sostituisce il valore della terna risultato *t1* con quello della terna *t*.
- `t.modifica(sel, a, b, c)`
  Operatore che modifica i valori dei campi della terna *t* specificati dal selettore *sel*, e lascia gli altri campi inalterati. Se il bit 0 del selettore vale 1, il primo campo della terna assume il valore *a*. Se il bit 1 del selettore vale 1, il secondo campo della terna assume il valore *b*. Infine, se il bit 2 del selettore vale 1, il terzo campo della terna assume il valore *c*.
- `t.somma(sel)`
  Operazione che ritorna la somma dei valori dei campi della terna *t* selezionati dal selettore *sel*.
- `cout << t`
  Operatore di uscita per il tipo *Terna*. L'uscita consiste nei valori dei campi che compongono la terna *t*, separati da virgole e racchiusi tra parentesi angolate. Esempio:

  ```
  <14, 9, 12>
  ```

- `~Terna()`
  Distruttore.

Una *terna multipla* è composta da un numero limitato di terne. Le terne che formano una terna multipla sono numerate a partire da 0. Le operazioni che possono essere effettuate su una terna multipla sono le seguenti:

- `TernaMultipla s;`
  Costruttore di default, che inizializza una terna multipla formata da una sola terna i cui campi hanno tutti valore unitario.
- `TernaMultipla s(n);`
  Costruttore che inizializza una terna multipla formata da *n* terne i cui campi hanno tutti valore unitario.
- `TernaMultipla s1(s);`
  Costruttore di copia, che inizializza la terna multipla *s1* col valore della terna multipla *s*.
- `s1 = s`
  Operatore di assegnamento, che sostituisce il valore della terna multipla risultato *s1* con quello della terna multipla *s*.
- `s.modifica(n, t)`
  Se il valore del parametro *n* è maggiore o uguale a 0, questa operazione sostituisce il valore della *n*-esima terna della terna multipla *s* con il valore della terna *t*. Se il valore del parametro *n* è minore di 0, l'operazione sostituisce il valore di ogni terna della terna multipla *s* col valore della terna *t*.
- `s.terna(n)`
  Se il valore del parametro *n* è maggiore o uguale a 0, questa operazione ritorna il valore

della *n*-esima terna della terna multipla *s*. Se il valore del parametro *n* è minore di 0, l'operazione ritorna una terna le cui componenti hanno valore pari alla somma delle corrispondenti componenti di tutte le terne della terna multipla *s*.

- `cout << s`
  Operatore di uscita per il tipo *TernaMultipla*. L'uscita ha la forma seguente:

  ```
  [3] <4, 2, 8> <12, 3, 18> <13, 2, 13>
  ```

  In questo esempio, la terna multipla è composta da tre terne. Il primo campo della terna 0 ha valore 4, il primo campo della terna 1 ha valore 12.

- `~TernaMultipla()`
  Distruttore.

Mediante il linguaggio C++, realizzare i tipi di dati astratti *Terna* e *TernaMultipla* definiti dalle precedenti specifiche. Individuare le eventuali situazioni di errore, e metterne in opera un corretto trattamento.

Soluzione

```
#include <iostream>
using namespace std;

class Terna {
   friend ostream& operator<<(ostream&, const Terna&);
   friend class TernaMultipla;
   int val[3];
public:
   Terna() {
      val[0] = val[1] = val[2] = 1; }
// Terna(const Terna&);
// Terna& operator=(const Terna&);
   Terna& modifica(unsigned int, int, int, int);
   int somma(unsigned int);
// ~Terna();
};

class TernaMultipla {
   friend ostream& operator<<(ostream&, const TernaMultipla&);
   Terna *tt;
   int dim;
public:
   TernaMultipla(int = 1);
   TernaMultipla(const TernaMultipla&);
   TernaMultipla& operator=(const TernaMultipla&);
   TernaMultipla& modifica(int, const Terna&);
   Terna terna(int) const;
   ~TernaMultipla() { delete[] tt; }
};

/* --------------- Classe Terna --------------- */

Terna& Terna::modifica(unsigned int sel, int a, int b, int c)
{
   if (sel & 1)
      val[0] = (a < 1) ? 1 : a;
   if (sel & 2)
      val[1] = (b < 1) ? 1 : b;
   if (sel & 4)
      val[2] = (c < 1) ? 1 : c;
   return *this;
}
```

```
int Terna::somma(unsigned int sel)
{
   int s = 0;
   if (sel & 1)
      s += val[0];
   if (sel & 2)
      s += val[1];
   if (sel & 4)
      s += val[2];
   return s;
}

ostream& operator<<(ostream& os, const Terna& t)
{
   return (os << '<' << t.val[0] << ", " << t.val[1] << ", "
        << t.val[2] << '>');
}

/* ------------ Classe TernaMultipla ------------- */

TernaMultipla::TernaMultipla(int n)
{
   dim = (n > 1) ? n : 1;
   tt = new Terna[dim];
}

TernaMultipla::TernaMultipla(const TernaMultipla& s)
{
   dim = s.dim;
   tt = new Terna[dim];
   for (int i = 0; i < dim; i++)
      tt[i] = s.tt[i];
}

TernaMultipla& TernaMultipla::operator=(const TernaMultipla& s)
{
   if (this != &s) {
      delete[] tt;
      dim = s.dim;
      tt = new Terna[dim];
      for (int i = 0; i < dim; i++)
         tt[i] = s.tt[i];
   }
   return *this;
}

TernaMultipla& TernaMultipla::modifica(int n, const Terna& t)
{
   n = (n > dim - 1) ? (dim - 1) : n;
   if (n >= 0)
      tt[n] = t;
   else
      for (int i = 0; i < dim; i++)
         tt[i] = t;
   return *this;
}

Terna TernaMultipla::terna(int n) const
{
   n = (n > dim - 1) ? (dim - 1) : n;
   if (n >= 0)
```

```
      return tt[n];
   Terna t(tt[0]);
   for (int i = 1; i < dim; i++) {
      t.val[0] += tt[i].val[0];
      t.val[1] += tt[i].val[1];
      t.val[2] += tt[i].val[2];
   }
   return t;
}

ostream& operator<<(ostream& os, const TernaMultipla& s)
{
   os << '[' << s.dim << "] ";
   for (int i = 0; i < s.dim; i++) {
      os << s.tt[i];
      if (i != s.dim - 1)
         os << ' ';
   }
   return os;
}
```

Esercizio n. 7 – Selettore

Un *selettore* è in grado di gestire un numero illimitato di richieste. Ciascuna richiesta è identificata da un numero intero maggiore di 0. Ad ogni dato istante, una delle richieste contenute nel selettore è marcata come richiesta *corrente*. Le operazioni che possono essere effettuate su un selettore sono le seguenti:

- `Selettore s;`
  Costruttore di default, che inizializza un selettore *s*. Inizialmente, il selettore non contiene richieste.
- `Selettore s1(s);`
  Costruttore di copia, che inizializza un selettore *s1* col valore del selettore *s*.
- `s1 = s`
  Operatore di assegnamento, che sostituisce il valore del selettore risultato *s1* con quello del selettore *s*.
- `s.inserisci(id)`
  Operazione che inserisce una nuova richiesta nel selettore *s*. La nuova richiesta ha identificatore *id*. Tale richiesta diviene la nuova richiesta corrente.
- `s.estrai()`
  Operazione che estrae la richiesta corrente dal selettore *s* e ne ritorna l'identificatore. A seguito dell'esecuzione di questa operazione, la richiesta il cui identificatore è più vicino (come valore intero) alla richiesta estratta diviene la nuova richiesta corrente (e pertanto, ad esempio, se nel selettore sono presenti quattro richieste aventi come identificatori 1, 3, 6, 7 e la richiesta corrente è la 3, viene estratta la richiesta 3 e la 1 diviene la nuova richiesta corrente).
- `!s`
  Operatore di negazione logica, che ritorna il numero di richieste presenti nel selettore *s*.
- `cout << s`
  Operatore di uscita per il tipo *Selettore*. L'uscita ha la forma seguente:

  ```
  1, [3], 6, 7
  ```

  In questo esempio, nel selettore sono presenti quattro richieste aventi come identificatori 1, 3, 6 e 7. La richiesta corrente è la 3.
- `~Selettore()`
  Distruttore.

Mediante il linguaggio C++, realizzare il tipo di dati astratti *Selettore* definito dalle precedenti specifiche. Utilizzare una rappresentazione del selettore a *lista bidirezionale*, nella quale ciascun elemento include un puntatore all'elemento precedente e un puntatore all'elemento successivo. Individuare le eventuali situazioni di errore, e metterne in opera un corretto trattamento.

Soluzione

```
#include <iostream>
using namespace std;

class Selettore {
   friend ostream& operator<<(ostream&, const Selettore&);
   struct elem {
      int req;
      elem *prec, *succ;
   };
   elem *primo, *corrente;
   void copia(const Selettore&);
```

```
   void elimina();
public:
   Selettore() { primo = corrente = NULL; }
   Selettore(const Selettore& s) { copia(s); }
   Selettore& operator=(const Selettore&);
   Selettore& inserisci(int);
   int estrai();
   int operator!() const;
   ~Selettore() { elimina(); }
};

void Selettore::copia(const Selettore& s)
{
   primo = corrente = NULL;
   if (s.primo != NULL) {
      primo = new elem;
      primo->req = s.primo->req;
      primo->prec = NULL;
      if (s.corrente == s.primo)
         corrente = primo;
      elem *aux = s.primo->succ, *ultimo = primo;
      while (aux != NULL) {
         ultimo->succ = new elem;
         ultimo->succ->prec = ultimo;
         ultimo = ultimo->succ;
         ultimo->req = aux->req;
         if (s.corrente == aux)
            corrente = ultimo;
         aux = aux->succ;
      }
      ultimo->succ = NULL;
   }
}

void Selettore::elimina()
{
   corrente = primo;
   while (primo != NULL) {
      primo = primo->succ;
      delete corrente;
      corrente = primo;
   }
}

Selettore& Selettore::operator=(const Selettore& s)
{
   if (this != &s) {
      elimina();
      copia(s);
   }
   return *this;
}

Selettore& Selettore::inserisci(int d)
{
   if (d > 0) {
      elem *precedente = primo, *successivo = primo;
      while (successivo != NULL && successivo->req < d) {
         precedente = successivo;
         successivo = successivo->succ;
      }
      corrente = new elem;
```

```
      corrente->req = d;
      corrente->prec = precedente;
      corrente->succ = successivo;
      if (successivo == primo) {
         primo = corrente;
         corrente->prec = NULL;
      }
      else
         precedente->succ = corrente;
      if (successivo != NULL)
         successivo->prec = corrente;
   }
   return *this;
}

int Selettore::estrai()
{
   if (corrente == NULL)
      return -1;
   elem *tmp = corrente;
   int d = tmp->req;
   if (corrente->prec == NULL) { // Estrazione in testa
      primo = corrente->succ;
      corrente = primo;
      if (corrente != NULL)
         corrente->prec = NULL;
   }
   else if (corrente->succ == NULL) { // Estrazione in fondo
      corrente = corrente->prec;
      corrente->succ = NULL;
   }
   else {
      corrente->prec->succ = corrente->succ;
      corrente->succ->prec = corrente->prec;
      corrente = ( (d - corrente->prec->req) > (corrente->succ->req - d) )
         ? corrente->succ : corrente->prec;
   }
   delete tmp;
   return d;
}

int Selettore::operator!() const
{
   int conta = 0;
   elem* aux = primo;
   while (aux != NULL) {
      aux = aux->succ;
      conta++;
   }
   return conta;
}

ostream& operator<<(ostream& os, const Selettore& s)
{
   if (s.primo != NULL) {
      if (s.corrente == s.primo)
         os << '[' << s.primo->req << ']';
      else
         os << s.primo->req;
      Selettore::elem *aux = s.primo->succ;
      while (aux != NULL) {
         if (s.corrente == aux)
```

```
            os << ", [" << aux->req << ']';
          else
            os << ", " << aux->req;
          aux = aux->succ;
        }
      }
      return os;
    }
```

Esercizio n. 8 – Scacchiera

Una *scacchiera* è formata da caselle aventi colori alterni, bianco e nero, ed organizzate in un uguale numero di righe e di colonne. Le righe e le colonne sono numerate a partire da 1. La casella di riga 1 e colonna 1 è bianca. Ciascuna casella può essere libera o occupata da una pedina colorata, ed i possibili colori delle pedine sono rosso, bianco e nero. Il colore di una casella è *compatibile* col colore di una data pedina se quella casella è bianca e la pedina è bianca o rossa, ovvero se quella casella è nera e la pedina è nera o rossa. Una pedina può occupare solo caselle di colore compatibile (e cioè, una pedina bianca può occupare solo una casella bianca, una pedina nera può occupare solo una casella nera, una pedina rossa può occupare qualsiasi casella, bianca o nera). Le operazioni che possono essere effettuate su una scacchiera sono le seguenti:

- `Scacchiera s;`
  Costruttore di default, che inizializza una scacchiera *s* formata da una sola casella di colore bianco. Inizialmente, tale casella è libera.
- `Scacchiera s(N);`
  Costruttore che inizializza una scacchiera *s* formata da *N* righe ed *N* colonne di caselle. Inizialmente, tutte le caselle della scacchiera sono libere.
- `Scacchiera s1(s);`
  Costruttore di copia, che inizializza la scacchiera *s1* col valore della scacchiera *s*.
- `s1 = s`
  Operatore di assegnamento, che sostituisce il valore della scacchiera risultato *s1* con quello della scacchiera *s*.
- `s += c`
  Operatore di somma e assegnamento, che inserisce nella scacchiera *s* una nuova pedina avente colore *c*. La pedina è inserita nella casella di *s* avente il più basso numero di riga e, a parità di numero di riga, il più basso numero di colonna tra le caselle libere aventi colore compatibile con *c*. Se la scacchiera *s* non ha caselle libere di colore compatibile con *c*, l'operazione lascia la scacchiera inalterata.
- `s *= c`
  Operatore di prodotto e assegnamento, che inserisce nella scacchiera *s* una nuova pedina avente colore *c*. La pedina è inserita nella casella di *s* avente il più alto numero di riga e, a parità di numero di riga, il più alto numero di colonna tra le caselle libere aventi colore compatibile con *c*. Se la scacchiera *s* non ha caselle libere di colore compatibile con *c*, l'operazione lascia la scacchiera inalterata.
- `s -= c`
  Operatore di sottrazione e assegnamento, che elimina dalla scacchiera *s* tutte le pedine aventi colore *c*.
- `s /= c`
  Operatore di divisione e assegnamento, che elimina dalla scacchiera *s* tutte le pedine aventi colore diverso da *c*.
- `cout << s`
  Operatore di uscita per il tipo *Scacchiera*. L'uscita ha la forma seguente:

  ```
  b:B n:R b:-
  n:N b:B n:-
  b:B n:R b:R
  ```

  In questo esempio, la scacchiera ha tre righe e tre colonne. La casella di riga 1 e colonna 1 è di colore bianco ed è occupata da una pedina bianca. La casella di riga 1 e colonna 2 è di colore nero ed è occupata da una pedina rossa. La casella di riga 1 e colonna 3 è di colore bianco ed è libera. La casella di riga 2 e colonna 1 è di colore nero ed occupata da una pedina nera.

- ~Scacchiera()
  Distruttore.

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

Soluzione

```
#include <iostream>
using namespace std;

enum colore { BIANCO, NERO, ROSSO };

class Scacchiera {
   friend ostream& operator<<(ostream&, const Scacchiera&);

   class casella {
      enum stato { LIBERA, OCCUPATA } st;
      colore col;
   public:
      casella() { st = LIBERA; }
      void occupa(colore c) { st = OCCUPATA; col = c; }
      void libera() { st = LIBERA; }
      int occupata() const { return st == OCCUPATA; }
      colore pedina() const { return col; }
   };

   int dim;
   casella *cc;
   int compatibile(int, colore);
public:
   Scacchiera(int = 1);
   Scacchiera(const Scacchiera&);
   Scacchiera& operator=(const Scacchiera&);
   Scacchiera& operator+=(colore c);
   Scacchiera& operator*=(colore c);
   Scacchiera& operator-=(colore c);
   Scacchiera& operator/=(colore c);
   ~Scacchiera() { delete[] cc; }
};

int Scacchiera::compatibile(int i, colore col)
{
/*
Il colore della casella i è nero se (i / dim + i % dim) è dispari,
ed è bianco se (i / dim + i % dim) è pari.
*/
   switch (col) {
      case ROSSO:
         return 1;
      case NERO:
         return ( (i / dim + i % dim) % 2);
      case BIANCO:
         return !( (i / dim + i % dim) % 2);
   }
}

Scacchiera::Scacchiera(int N)
{
   dim = N > 0 ? N : 1;
```

```
   cc = new casella[dim * dim];
}

Scacchiera::Scacchiera(const Scacchiera& s)
{
   dim = s.dim;
   int lun = dim * dim;
   cc = new casella[lun];
   for (int i = 0; i < lun; i++)
      cc[i] = s.cc[i];
}

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

Scacchiera& Scacchiera::operator+=(colore c)
{
   for (int i = 0; i < dim * dim; i++)
      if (compatibile(i, c) && !cc[i].occupata()) {
         cc[i].occupa(c);
         break;
      }
   return *this;
}

Scacchiera& Scacchiera::operator*=(colore c)
{
   for (int i = dim * dim - 1; i >= 0; i--)
      if (compatibile(i, c) && !cc[i].occupata()) {
         cc[i].occupa(c);
         break;
      }
   return *this;
}

Scacchiera& Scacchiera::operator-=(colore c)
{
   for (int i = 0; i < dim * dim; i++)
      if (cc[i].occupata() && cc[i].pedina() == c)
         cc[i].libera();
   return *this;
}

Scacchiera& Scacchiera::operator/=(colore c)
{
   for (int i = 0; i < dim * dim; i++)
      if (cc[i].occupata() && cc[i].pedina() != c)
         cc[i].libera();
   return *this;
}

ostream& operator<<(ostream& os, const Scacchiera& s)
```

```
{
   for (int i = 0; i < s.dim; i++) {
      for (int j = 0; j < s.dim; j++) {
         os << ((i + j) % 2 ? "n:" : "b:");
         if (s.cc[i * s.dim + j].occupata())
            switch ( s.cc[i * s.dim + j].pedina() ) {
               case BIANCO:
                  os << 'B';
                  break;
               case NERO:
                  os << 'N';
                  break;
               case ROSSO:
                  os << 'R';
            }
         else
            os << '-';
         if (j < s.dim - 1)
            os << ' ';
      }
      if (i < s.dim - 1)
         os << '\n';
   }
   return os;
}
```

Esercizio n. 9 – Mazzo da gioco

Un *mazzo da gioco* contiene un numero illimitato di carte aventi come possibili semi: picche, cuori, fiori, quadri; e come possibili valori: da 1 a 10. In un dato mazzo, ciascuna carta può essere presente 0 o più volte, e pertanto il mazzo può contenere due o più carte identiche. Le operazioni che possono essere effettuate su un mazzo da gioco sono le seguenti:

- `MazzoDaGioco m;`
  Costruttore di default, che inizializza un mazzo da gioco *m*. Inizialmente, il mazzo non contiene carte.
- `MazzoDaGioco m(s, v);`
  Costruttore che inizializza un mazzo da gioco *m* formato da una sola carta. Tale carta ha seme *s* e valore *v*.
- `MazzoDaGioco m(s);`
  Costruttore che inizializza un mazzo da gioco *m* formato da dieci carte di seme *s* e valori tutti diversi tra loro.
- `MazzoDaGioco m(v);`
  Costruttore che inizializza un mazzo da gioco *m* formato da quattro carte di valore *v* e semi tutti diversi tra loro.
- `MazzoDaGioco m1(m);`
  Costruttore di copia, che inizializza il mazzo da gioco *m1* col valore del mazzo da gioco *m*.
- `m1 = m`
  Operatore di assegnamento, che sostituisce il valore del mazzo da gioco risultato *m1* con quello del mazzo da gioco *m*.
- `m += m1`
  Operatore di somma e assegnamento, che aggiunge al mazzo da gioco *m* le carte contenute nel mazzo da gioco *m1*. L'operazione lascia *m1* inalterato.
- `m += s`
  Operatore di somma e assegnamento, che aggiunge al mazzo da gioco *m* dieci carte di seme *s* e valori diversi.
- `m += v`
  Operatore di somma e assegnamento, che aggiunge al mazzo da gioco *m* quattro carte di valore *v* e semi diversi.
- `m -= m1`
  Operatore di sottrazione e assegnamento, che toglie dal mazzo da gioco *m* le carte contenute nel mazzo da gioco *m1*. L'operazione lascia *m1* inalterato.
- `m -= s`
  Operatore di sottrazione e assegnamento, che toglie dal mazzo da gioco *m* una carta di seme *s* per ciascun valore (cioè toglie al più 10 carte).
- `m -= v`
  Operatore di sottrazione e assegnamento, che toglie dal mazzo da gioco *m* una carta di valore *v* per ciascun seme (cioè toglie al più 4 carte).
- `m %= s`
  Operatore di modulo e assegnamento, che toglie dal mazzo da gioco *m* tutte le carte di seme diverso da *s*
- `m %= v`
  Operatore di modulo e assegnamento, che toglie dal mazzo da gioco *m* tutte le carte di valore diverso da *v*.
- `cout << m`
  Operatore di uscita per il tipo *MazzoDaGioco*. L'uscita consiste nel numero di carte che formano il mazzo da gioco *m*.

- ~MazzoDaGioco()
  Distruttore.

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

Soluzione

```
#include <iostream>
using namespace std;

enum seme { PICCHE, CUORI, QUADRI, FIORI };

class MazzoDaGioco {
   friend ostream& operator<<(ostream&, const MazzoDaGioco&);
   int mazzo[10][4];
   int quante() const;
public:
   MazzoDaGioco();
   MazzoDaGioco(seme);
   MazzoDaGioco(int);
   MazzoDaGioco(seme, int);
// MazzoDaGioco(const MazzoDaGioco&);
// MazzoDaGioco& operator=(const MazzoDaGioco&);
   MazzoDaGioco& operator+=(const MazzoDaGioco&);
// MazzoDaGioco& operator+=(seme s);
// MazzoDaGioco& operator+=(int v);
   MazzoDaGioco& operator-=(const MazzoDaGioco&);
// MazzoDaGioco& operator-=(seme s);
// MazzoDaGioco& operator-=(int v);
   MazzoDaGioco& operator%=(seme);
   MazzoDaGioco& operator%=(int);
// ~MazzoDaGioco();
};

int MazzoDaGioco::quante() const
{
   int somma = 0;
   for (int i = 0; i < 10; i++)
      for (int j = PICCHE; j <= FIORI; j++)
         somma += mazzo[i][j];
   return somma;
}

MazzoDaGioco::MazzoDaGioco()
{
   for (int i = 0; i < 10; i++)
      for (int j = PICCHE; j <= FIORI; j++)
         mazzo[i][j] = 0;
}

MazzoDaGioco::MazzoDaGioco(seme s)
{
   for (int i = 0; i < 10; i++)
      for (int j = PICCHE; j <= FIORI; j++)
         mazzo[i][j] = (j == s) ? 1 : 0;
}

MazzoDaGioco::MazzoDaGioco(int v)
{
```

```
   for (int i = 0; i < 10; i++)
      for (int j = PICCHE; j <= FIORI; j++)
         mazzo[i][j] = (i == v - 1) ? 1 : 0;
}

MazzoDaGioco::MazzoDaGioco(seme s, int v)
{
   for (int i = 0; i < 10; i++)
      for (int j = PICCHE; j <= FIORI; j++)
         mazzo[i][j] = 0;
   if (v >= 1 && v <= 10)
      mazzo[v - 1][s] = 1;
}

MazzoDaGioco& MazzoDaGioco::operator+=(const MazzoDaGioco& m)
{
   for (int i = 0; i < 10; i++)
      for (int j = PICCHE; j <= FIORI; j++)
         mazzo[i][j] += m.mazzo[i][j];
   return *this;
}

MazzoDaGioco& MazzoDaGioco::operator-=(const MazzoDaGioco& m)
{
   for (int i = 0; i < 10; i++)
      for (int j = PICCHE; j <= FIORI; j++)
         if (mazzo[i][j] >= m.mazzo[i][j])
            mazzo[i][j] -= m.mazzo[i][j];
         else
            mazzo[i][j] = 0;
   return *this;
}

MazzoDaGioco& MazzoDaGioco::operator%=(seme s)
{
   for (int i = 0; i < 10; i++)
      for (int j = PICCHE; j <= FIORI; j++)
         if (j != s)
            mazzo[i][j] = 0;
   return *this;
}

MazzoDaGioco& MazzoDaGioco::operator%=(int v)
{
   if (v >= 1 && v <= 10)
      for (int i = 0; i < 10; i++)
         for (int j = PICCHE; j <= FIORI; j++)
            if (i != v - 1)
               mazzo[i][j] = 0;
   return *this;
}

ostream& operator<<(ostream& os, const MazzoDaGioco& m)
{
   return os << m.quante();
}
```

Esercizio n. 10 – Righello

Un *righello* è formato da caselle numerate a partire da 0. Ciascuna casella ha un valore intero maggiore di 0 e può essere libera o occupata da una pedina. Le pedine hanno un valore intero maggiore di 0. Ogni pedina può occupare solo caselle aventi valore maggiore o uguale al proprio valore. Le operazioni che possono essere effettuate su un righello sono le seguenti:

- `Righello s;`
  Costruttore di default, che inizializza un righello *s* formato da una sola casella. Inizialmente, tale casella è libera ed ha valore 1.
- `Righello s(N);`
  Costruttore che inizializza un righello *s* formato da *N* caselle. Inizialmente, tutte le caselle sono libere ed hanno valore 1.
- `Righello s1(s);`
  Costruttore di copia, che inizializza il righello *s1* col valore del righello *s*.
- `s1 = s`
  Operatore di assegnamento, che sostituisce il valore del righello risultato *s1* con quello del righello *s*.
- `s += n`
  Operatore di somma e assegnamento, che occupa con una pedina di valore *n* la casella del righello *s* col più alto numero d'ordine tra le caselle libere aventi valore maggiore o uguale a quello della pedina.
- `s -= n`
  Operatore di sottrazione e assegnamento, che elimina dal righello *s* una pedina di valore *n*. Se due o più pedine hanno valore *n*, viene eliminata quella posizionata sulla casella col più alto numero d'ordine.
- `s.valore(i, n)`
  Operazione che attribuisce il valore *n* alla casella *i* del righello *s*. Tale casella deve essere libera.
- `s *= n`
  Operatore di prodotto e assegnamento, che occupa con pedine di valore *n* tutte le casella libere del righello *s* aventi valore maggiore o uguale ad *n*.
- `s /= n`
  Operatore di divisione e assegnamento, che elimina dal righello *s* tutte le pedine aventi valore *n*.
- `cout << s`
  Operatore di uscita per il tipo *Righello*. L'uscita ha la forma seguente:

  (12) 4, (13) 9, (1), (7) 7

  In questo esempio, il righello *s* ha quattro caselle. La prima casella ha valore 12 ed è occupata da una pedina avente valore 4, la terza casella ha valore 1 ed è libera, l'ultima casella ha valore 7 ed è occupata da una pedina avente valore 7.
- `~Righello()`
  Distruttore.

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

Soluzione

```
#include <iostream>
```

```
using namespace std;

class Righello {
   friend ostream& operator<<(ostream&, const Righello&);
   struct casella {
      int valore;  // Valore della casella.
      int pedina;  // Valore della pedina che occupa la casella.
// Se pedina vale 0, la casella è libera.
   };
   casella *rig;
   int dim;
public:
   Righello(int = 1);
   Righello(const Righello&);
   Righello& operator=(const Righello&);
   Righello& operator+=(int);
   Righello& operator-=(int);
   Righello& valore(int, int);
   Righello& operator*=(int);
   Righello& operator/=(int);
   ~Righello() { delete[] rig; }
};

Righello::Righello(int n)
{
   dim = (n > 0) ? n : 1;
   rig = new casella[dim];
   casella iniziale = {1, 0};
   for (int i = 0; i < dim; i++)
      rig[i] = iniziale;
}

Righello::Righello(const Righello& s)
{
   dim = s.dim;
   rig = new casella[dim];
   for (int i = 0; i < dim; i++)
      rig[i] = s.rig[i];
}

Righello& Righello::operator=(const Righello& s)
{
   if (this != &s) {
      if (dim != s.dim) {
         delete[] rig;
         dim = s.dim;
         rig = new casella[dim];
      }
      for (int i = 0; i < dim; i++)
         rig[i] = s.rig[i];
   }
   return *this;
}

Righello& Righello::operator+=(int n)
{
   if (n > 0)
      for (int i = dim - 1; i >= 0; i--)
         if (!rig[i].pedina && rig[i].valore >= n) {
            rig[i].pedina = n;
            break;
         }
```

```
   return *this;
}


Righello& Righello::operator-=(int n)
{
   if (n > 0)
      for (int i = dim - 1; i >= 0; i--)
         if (rig[i].pedina == n) {
            rig[i].pedina = 0;
            break;
         }
   return *this;
}


Righello& Righello::valore(int i, int n)
{
   if (n > 0 && i >= 0 && i < dim && !rig[i].pedina)
      rig[i].valore = n;
   return *this;
}


Righello& Righello::operator*=(int n)
{
   if (n > 0)
      for (int i = 0; i < dim; i++)
         if (rig[i].valore >= n && !rig[i].pedina)
            rig[i].pedina = n;
   return *this;
}


Righello& Righello::operator/=(int n)
{
   if (n > 0)
      for (int i = 0; i < dim; i++)
         if (rig[i].pedina == n)
            rig[i].pedina = 0;
   return *this;
}


ostream& operator<<(ostream& os, const Righello& s)
{
   for (int i = 0; i < s.dim; i++) {
      os << '(' << s.rig[i].valore << ')';
      if (s.rig[i].pedina != 0)
         os << s.rig[i].pedina;
      if (i < s.dim - 1)
         os << ", ";
   }
   return os;
}
```

Esercizio n. 11 – Tiro a segno

Un *tiro a segno* è formato da al più 26 pile di barattoli, contro le quali vengono lanciate palle di gomma. Le pile di barattoli che formano il tiro a segno sono mantenute *equilibrate* nel numero di barattoli, e cioè, le prime pile, più alte, hanno un solo barattolo in più rispetto alle ultime, più basse (pertanto, ad esempio, in un tiro a segno con 5 pile e 23 barattoli, le prime tre pile sono formate da 5 barattoli e le ultime due sono formate da 4 barattoli). Le pile sono individuate da lettere maiuscole dell'alfabeto, a partire dalla lettera A. I barattoli di una pila sono numerati a partire dalla posizione più in basso (il barattolo 1). Le operazioni che possono essere effettuate su un tiro a segno sono le seguenti:

- `TiroASegno m;`
  Costruttore di default, che inizializza un tiro a segno *m* formato da una pila contenente un solo barattolo.
- `TiroASegno m(n);`
  Costruttore che inizializza un tiro a segno *m* formato da una pila contenente *n* barattoli.
- `TiroASegno m(n, p);`
  Costruttore che inizializza un tiro a segno *m* formato da *p* pile e *n* barattoli. Le pile sono equilibrate.
- `TiroASegno m1(m);`
  Costruttore di copia, che inizializza il tiro a segno *m1* col valore del tiro a segno *m*.
- `m1 = m`
  Operatore di assegnamento, che sostituisce il valore del tiro a segno risultato *m1* con quello del tiro a segno *m*.
- `m.palla(p, b)`
  Operazione che produce il lancio di una palla contro il barattolo *b* della pila *p* del tiro a segno *m*. Tale barattolo cade insieme a tutti i barattoli nelle posizioni superiori della medesima pila. Successivamente, le pile che formano il tiro a segno vengono riequilibrate utilizzando i barattoli residui (cioè non caduti a causa del lancio).
- `m *= p`
  Operatore di prodotto e assegnamento, che aggiunge *p* pile di barattoli al tiro a segno *m*. Il numero complessivo di barattoli nel tiro a segno resta inalterato. Le pile di barattoli, incluse le nuove pile, vengono riequilibrate.
- `m /= p`
  Operatore di divisione e assegnamento, che elimina *p* pile di barattoli dal tiro a segno *m*. Il numero complessivo di barattoli nel tiro a segno resta inalterato. Le pile di barattoli rimanenti vengono riequilibrate.
- `m % p`
  Operatore di modulo, che ritorna il numero di barattoli che formano la pila *p* del tiro a segno *m.*
- `cout << m`
  Operatore di uscita per il tipo *TiroASegno*. L'uscita ha la forma seguente:

  ```
  <5, 5, 5, 4, 4>
  ```

  In questo esempio, il tiro a segno è formato da 5 pile e un totale di 23 barattoli. Le pile A, B e C sono formate da 5 barattoli, le pile D e E sono formate da 4 barattoli.
- `~TiroASegno()`
  Distruttore.

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

Soluzione

```
#include <iostream>
using namespace std;

class TiroASegno {
   friend ostream& operator<<(ostream&, const TiroASegno&);
   int numPile, numBarattoli;
public:
   TiroASegno(int = 1,int = 1);
// TiroASegno(const TiroASegno&);
// TiroASegno& operator=(const TiroASegno&);
   TiroASegno& palla(char, int);
   TiroASegno& operator*=(int);
   TiroASegno& operator/=(int);
   int operator%(char) const;
// ~TiroASegno();
};

TiroASegno::TiroASegno(int n, int p)
{
   numPile = p < 1 ? 1 : (p > 26 ? 26 : p);
   numBarattoli = n < 1 ? 1 : n;
}

TiroASegno& TiroASegno::palla(char p, int b)
{
// La quantità numBarattoli / numPile rappresenta il numero di barattoli
// in una pila bassa.
// La quantità numBarattoli % numPile rappresenta il numero di pile alte.
   if (p >= 'A' && p <= 'A' + numPile - 1) {
      int numBarattoliInPila = numBarattoli / numPile;
      if (p <= 'A' + (numBarattoli % numPile) - 1) // è una pila alta
         numBarattoliInPila++;
      if (b > 0 && b <= numBarattoliInPila)
         numBarattoli -= numBarattoliInPila - b + 1;
   }
   return *this;
}

TiroASegno& TiroASegno::operator*=(int p)
{
   if (p > 0) {
      numPile += p;
      numPile = (numPile > 26) ? 26 : numPile;
   }
   return *this;
}

TiroASegno& TiroASegno::operator/=(int p)
{
   if (p > 0) {
      numPile -= p;
      numPile = (numPile < 1) ? 1 : numPile;
   }
   return *this;
}

int TiroASegno::operator%(char p) const
{
   if (p < 'A')
      return -1;
```

```
   if (p <= 'A' + (numBarattoli % numPile) - 1) // è una pila alta
      return numBarattoli / numPile + 1;
   if (p <= 'A' - 1 + numPile) // è una pila bassa
      return numBarattoli / numPile;
   return -1;
}

ostream& operator<<(ostream& os, const TiroASegno& m)
{
   int numPileAlte = m.numBarattoli % m.numPile;
   int numBarattoliBasse = m.numBarattoli / m.numPile;
   os << '<';
   for (int i = 0; i < m.numPile; i++) {
      if (i < numPileAlte)
         os << numBarattoliBasse + 1;
      else
         os << numBarattoliBasse;
      if (i < m.numPile - 1)
         os << ", ";
   }
   os << '>';
   return os;
}
```

Esercizio n. 12 – Lista partizionata

Una *lista partizionata* è formata da un numero illimitato di elementi aventi valore intero. La lista è divisa in sottoliste numerate a partire da 0. Le operazioni che possono essere effettuate su una lista partizionata sono le seguenti:

- `ListaPartizionata s;`
  Costruttore di default, che inizializza una lista partizionata *s* formata da una sola sottolista. Inizialmente, la lista partizionata è vuota.
- `ListaPartizionata s(n);`
  Costruttore che inizializza una lista partizionata *s* formata da *n* sottoliste. Inizialmente, la lista partizionata è vuota.
- `ListaPartizionata s1(s);`
  Costruttore di copia, che inizializza la lista partizionata *s1* col valore della lista partizionata *s*.
- `s1 = s`
  Operatore di assegnamento, che sostituisce il valore della lista partizionata risultato *s1* con quello della lista partizionata *s*.
- `s.inserisci(v)`
  Operazione che aggiunge alla lista partizionata *s* un nuovo elemento avente valore *v*. Il nuovo elemento viene inserito in testa alla sottolista di *s* che contiene il minor numero di elementi. L'operazione ritorna il numero d'ordine di tale sottolista.
- `s.estrai(i, v)`
  Operazione che estrae l'elemento in testa alla sottolista *i* della lista partizionata *s*, e ritorna il valore *v* di tale elemento. L'operazione fallisce se la sottolista *i* è vuota. L'operazione ritorna un'indicazione di successo (1) o fallimento (0).
- `s % i`
  Operatore di modulo, che ritorna il numero di elementi che formano la sottolista *i* della lista partizionata *s.*
- `!s`
  Operatore di negazione logica, che ritorna il numero totale di elementi che formano la lista partizionata *s*.
- `cout << s`
  Operatore di uscita per il tipo *ListaPartizionata*. L'uscita ha la forma seguente:

  ```
  <5, 4, 0, 2, 3>
  ```

  In questo esempio, la lista partizionata *s* è formata da cinque sottoliste. La sottolista 0 contiene 5 elementi, la sottolista 3 contiene 2 elementi.
- `~ListaPartizionata()`
  Distruttore.

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

Soluzione

```
#include <iostream>
using namespace std;

class ListaPartizionata {

   class Lista {
      struct elem {
```

```
        int val;
        elem *succ;
      };
      elem *testa;
      void elimina();
      void copia(const Lista&);
   public:
      Lista() { testa = NULL; }
      Lista(const Lista& s) { copia(s); }
      Lista& operator=(const Lista&);
      void inserisci(int);
      int estrai(int&);
      int operator!() const;
      ~Lista() { elimina(); }
   };

   friend ostream& operator<<(ostream&, const ListaPartizionata&);
   Lista *aa;
   int dim;
public:
   ListaPartizionata(int = 1);
   ListaPartizionata(const ListaPartizionata&);
   ListaPartizionata& operator=(const ListaPartizionata&);
   int inserisci(int);
   int estrai(int, int&);
   int operator%(int) const;
   int operator!() const;
   ~ListaPartizionata() { delete[] aa; }
};

/* -------------------- Classe Lista -------------------- */

void ListaPartizionata::Lista::elimina()
{
   elem *tmp = testa;
   while (testa != NULL) {
      testa = testa->succ;
      delete tmp;
      tmp = testa;
   }
}

void ListaPartizionata::Lista::copia(const Lista& s)
{
   testa = NULL;
   if (s.testa != NULL) {
      testa = new elem;
      testa->val = s.testa->val;
      elem *tmp = testa, *aux = s.testa->succ;
      while (aux != NULL) {
         tmp->succ = new elem;
         tmp = tmp->succ;
         tmp->val = aux->val;
         aux = aux->succ;
      }
      tmp->succ = NULL;
   }
}

ListaPartizionata::Lista&
ListaPartizionata::Lista::operator=(const Lista& s)
{
```

```
   if (this != &s) {
      elimina();
      copia(s);
   }
   return *this;
}

void ListaPartizionata::Lista::inserisci(int v)
// Inserzione in testa
{
   elem *tmp = new elem;
   tmp->val = v;
   tmp->succ = testa;
   testa = tmp;
}

int ListaPartizionata::Lista::estrai(int& v)
{
   if (testa != NULL) {
      elem *tmp = testa;
      testa = testa->succ;
      v = tmp->val;
      delete tmp;
      return 1;
   }
   return 0;
}

int ListaPartizionata::Lista::operator!() const
{
   int conta = 0;
   for (elem *tmp = testa; tmp != NULL; tmp = tmp->succ)
      conta++;
   return conta;
}

/* ------------- Classe ListaPartizionata ------------- */

ListaPartizionata::ListaPartizionata(int n)
{
   dim = (n > 0) ? n : 1;
   aa = new Lista[dim];
}

ListaPartizionata::ListaPartizionata(const ListaPartizionata& s)
{
   dim = s.dim;
   aa = new Lista[dim];
   for (int i = 0; i < dim; i++)
      aa[i] = s.aa[i];
// chiamata di ListaPartizionata::Lista::operator=()
}

ListaPartizionata&
ListaPartizionata::operator=(const ListaPartizionata& s)
{
   if (this != &s) {
      if (dim != s.dim) {
         dim = s.dim;
         delete[] aa;
         aa = new Lista[dim];
      }
```

```
      for (int i = 0; i < dim; i++)
         aa[i] = s.aa[i];
   }
   return *this;
}

int ListaPartizionata::inserisci(int v)
{
   int j = 0;
   for (int i = 1; i < dim; i++)
      if (!aa[i] < !aa[j])
         j = i;
   aa[j].inserisci(v);
   return j;
}

int ListaPartizionata::estrai(int i, int& v)
{
   if (i >= 0 && i < dim)
      return aa[i].estrai(v);
   return 0;
}

int ListaPartizionata::operator%(int i) const
{
   if (i >= 0 && i < dim)
      return !aa[i];
   return -1;
}

int ListaPartizionata::operator!() const
{
   int somma = 0;
   for (int i = 0; i < dim; i++)
      somma += !aa[i];
   return somma;
}

ostream& operator<<(ostream& os, const ListaPartizionata& s)
{
   os << '<';
   for (int i = 0; i < s.dim; i++) {
      os << (s % i);
      if (i != s.dim - 1)
         os << ", ";
   }
   os << '>';
   return os;
}
```

Esercizio n. 13 – Buffer

Un *buffer* è in grado di contenere un numero limitato di elementi gestiti secondo la strategia FIFO (*first in, first out*) in base alla quale viene estratto dal buffer l'elemento che è stato inserito da più tempo. Ciascun elemento ha un valore intero ed un livello di priorità alta o bassa. La *capacità* del buffer è il numero massimo di elementi della stessa priorità che il buffer è in grado di contenere (pertanto, ad esempio, un buffer di capacità 5 è in grado di contenere al più 5 elementi ad alta priorità ed inoltre al più 5 elementi a bassa priorità, per un totale di al più 10 elementi). Le operazioni che possono essere effettuate su un buffer sono le seguenti:

- `Buffer s;`
  Costruttore di default, che inizializza buffer *s* avente capacità unitaria. Inizialmente il buffer è vuoto.
- `Buffer s(c);`
  Costruttore che inizializza un buffer *s* avente capacità *c*. Inizialmente il buffer è vuoto.
- `Buffer s1(s);`
  Costruttore di copia, che inizializza il buffer *s1* col valore del buffer *s*.
- `s1 = s`
  Operatore di assegnamento, che sostituisce il valore del buffer risultato *s1* con quello del buffer *s*.
- `s.inserisci(v, p)`
  Operazione che inserisce nel buffer *s* un elemento avente priorità *p* e valore *v*. L'operazione fallisce in caso di buffer pieno per la priorità *p*. L'operazione ritorna un'indicazione di successo (1) o fallimento (0).
- `s.inserisci(v)`
  Operazione che inserisce nel buffer *s* un elemento avente priorità alta e valore v. Nel caso di buffer pieno per la priorità alta, l'elemento viene inserito a priorità bassa. L'operazione fallisce in caso di buffer pieno per ambedue le priorità. L'operazione ritorna un'indicazione di successo (1) o fallimento (0).
- `s.estrai(v, p)`
  Operazione che estrae dal buffer *s* un elemento avente priorità *p* e ne ritorna il valore *v*. L'operazione fallisce in caso di buffer vuoto per la priorità *p*. L'operazione ritorna un'indicazione di successo (1) o fallimento (0).
- `s.estrai(v)`
  Operazione che estrae dal buffer *s* un elemento avente priorità alta e ne ritorna il valore *v*. Nel caso di buffer vuoto per la priorità alta, viene estratto un elemento a priorità bassa. L'operazione fallisce in caso di buffer vuoto per ambedue le priorità. L'operazione ritorna un'indicazione di successo (1) o fallimento (0).
- `s % p`
  Operatore di modulo, che ritorna il numero di elementi aventi priorità *p* contenuti nel buffer *s.*
- `!s`
  Operatore di negazione logica, che ritorna il numero totale di elementi contenuti nel buffer *s*.
- `~Buffer()`
  Distruttore.

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

Soluzione

```
#include <iostream>
using namespace std;

enum prio { ALTA, BASSA };

class Buffer {

   class BufferCircolare {
      int prox_in, prox_out, num_elem, dim_buffer;
      int *buf;
   public:
      BufferCircolare(int = 1);
      BufferCircolare(const BufferCircolare& b);
      BufferCircolare& operator=(const BufferCircolare& b);
      int estrai(int &n);
      int inserisci(int n);
      int operator!() const { return num_elem; }
      ~BufferCircolare() { delete[] buf; }
   };

   BufferCircolare bb[2];
public:
   Buffer(int c = 1) {
      if (c > 1) {
         bb[ALTA]  = BufferCircolare(c);
         bb[BASSA] = BufferCircolare(c);
      }
   }
// Buffer(const Buffer&);
// Buffer& operator=(const Buffer&);
   int inserisci(int v, prio p) { return bb[p].inserisci(v); }
   int inserisci(int v) {
      return (bb[ALTA].inserisci(v) || bb[BASSA].inserisci(v)); }
                                                  // OR in cortocircuito
   int estrai(int& v, prio p) { return bb[p].estrai(v); }
   int estrai(int &v) {
      return ( bb[ALTA].estrai(v) || bb[BASSA].estrai(v) ); }
                                                  // OR in cortocircuito
   int operator%(prio p) const { return !bb[p]; }
   int operator!() const { return !bb[ALTA] + !bb[BASSA]; }
// ~Buffer();
};

Buffer::BufferCircolare::BufferCircolare(int n)
{
   prox_in = prox_out = num_elem = 0;
   dim_buffer = n;
   buf = new int[dim_buffer];
}

Buffer::BufferCircolare::BufferCircolare(const BufferCircolare& b)
{
   prox_in = b.prox_in;
   prox_out = b.prox_out;
   num_elem = b.num_elem;
   dim_buffer = b.dim_buffer;
   buf = new int[dim_buffer];
   for (int i = 0; i < dim_buffer; i++)
      buf[i] = b.buf[i];
}
```

```
Buffer::BufferCircolare&
Buffer::BufferCircolare::operator=(const BufferCircolare& b)
{
   if (this != &b) {
      prox_in = b.prox_in;
      prox_out = b.prox_out;
      num_elem = b.num_elem;
      if (dim_buffer != b.dim_buffer) {
         delete[] buf;
         dim_buffer = b.dim_buffer;
         buf = new int[dim_buffer];
      }
      for (int i = 0; i < dim_buffer; i++)
         buf[i] = b.buf[i];
   }
   return *this;
}

int Buffer::BufferCircolare::estrai(int &n)
{
   if (num_elem > 0) {
      n = buf[prox_out];
      prox_out++;
      if (prox_out == dim_buffer)
         prox_out = 0;
      num_elem--;
      return 1;
   }
   return 0;
}

int Buffer::BufferCircolare::inserisci(int n)
{
   if (num_elem < dim_buffer) {
      buf[prox_in] = n;
      prox_in++;
      if (prox_in == dim_buffer)
         prox_in = 0;
      num_elem++;
      return 1;
   }
   return 0;
}
```

Esercizio n. 14 – Torre

Una *torre* è formata da al più N dischi sovrapposti secondo diametri decrescenti. La quantità N è una costante globale intera positiva. Il diametro di un disco è espresso da un numero intero positivo. Nella torre possono esistere due o più dischi aventi lo stesso diametro. Le operazioni che possono essere effettuate su una torre sono le seguenti:

- `Torre s;`
  Costruttore di default, che inizializza una torre *s* formata da un solo disco di dimensione unitaria.
- `Torre s(d);`
  Costruttore che inizializza una torre *s* formata da un solo disco di dimensione *d*.
- `Torre s1(s);`
  Costruttore di copia, che inizializza la torre *s1* col valore della torre *s*.
- `s = s1`
  Operatore di assegnamento, che sostituisce il valore della torre risultato *s* con quello della torre *s1*.
- `s1 + s2`
  Operatore di addizione che ritorna la torre che si ottiene aggiungendo ai dischi della torre *s1* i dischi della torre *s2*.
- `s1 += s2`
  Operatore di addizione e assegnamento che aggiunge alla torre *s1* i dischi della torre *s2*.
- `s1 - s2`
  Operatore di sottrazione che ritorna la torre che si ottiene togliendo dalla torre *s1* i dischi della torre *s2*.
- `s1 -= s2`
  Operatore di sottrazione e assegnamento che toglie dalla torre *s1* i dischi della torre *s2*.
- `!s`
  Operatore di negazione logica, che ritorna il numero totale di dischi che formano la torre *s*.
- `s % n`
  Operatore di modulo, che ritorna il numero di dischi della torre *s* che hanno diametro *n*.
- `cin >> s`
  Operatore d'ingresso per il tipo *Torre*. L'ingresso consiste in un valore intero. L'operatore aggiunge alla torre *s* un disco di dimensione pari a tale valore intero.
- `cout << s`
  Operatore di uscita per il tipo *Torre*. L'uscita ha la forma

  ```
  *
  **
  ******
  ******
  *******
  ```

  In questo esempio, la torre è formata da 5 dischi aventi diametro pari a 1, 2, 6, 6 e 7, rispettivamente.
- `~Torre()`
  Distruttore.

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

Soluzione

```
const int N = 10;  // ad esempio

#include <iostream>
using namespace std;

class Torre {
   friend Torre operator+(const Torre&, const Torre&);
   friend Torre operator-(const Torre&, const Torre&);
   friend ostream& operator<<(ostream&, const Torre&);
   int diametri[N];
// Il vettore è mantenuto ordinato secondi diametri crescenti.
// Il valore 0 indica un elemento libero.
// Gli elementi liberi sono posizionati agli indici più bassi.
   void ordina();
public:
   Torre(int = 1);
// Torre(const Torre&);
// Torre& operator=(const Torre&);
   Torre& operator+=(const Torre&);
   Torre& operator-=(const Torre&);
   Torre& operator*=(const Torre&);
   Torre& operator/=(const Torre&);
   int operator!() const;
   int operator%(int) const;
// ~Torre();
};

void Torre::ordina() // Ordinamento in senso crescente
{
   int min, tmp;
   for (int i = 0; i < N - 1; i++) {
      min = i;
      for (int j = i + 1; j < N; j++)
         if (diametri[j] < diametri[min])
            min = j;
      tmp = diametri[min];
      diametri[min] = diametri[i];
      diametri[i] = tmp;
   }
}

Torre::Torre(int d)
{
   diametri[N - 1] = (d < 1) ? 1 : d;
// Inizialmente tutti gli elementi del vettore sono liberi, e pertanto
// hanno valore 0, tranne l'ultimo elemento che vale d.
   for (int i = 0; i < N - 1; i++)
      diametri[i] = 0;
}

Torre& Torre::operator+=(const Torre& s2)
{
   if ( !(*this) + !s2 > N )
      return *this;
   int i = 0, j = N - 1;
   while (diametri[i] == 0 && s2.diametri[j] > 0 && i < N && j >= 0)
      diametri[i++] = s2.diametri[j--];
   ordina();
   return *this;
}
```

```
Torre operator+(const Torre& s1, const Torre& s2)
{
   Torre tt(s1);
   tt += s2;
   return tt;
}

Torre& Torre::operator-=(const Torre& s2)
{
// Per ogni disco in s2, elimino uno disco di pari diametro, se esiste.
   for (int i = N - 1; i >= 0 && s2.diametri[i] > 0; i--) {
      for (int j = N - 1; j >= 0; j--)
         if (s2.diametri[i] == diametri[j]) {
            diametri[j] = 0;
            break;
         }
   }
   ordina();
   return *this;
}

Torre operator-(const Torre& s1, const Torre& s2)
{
   Torre tt(s1);
   tt -= s2;
   return tt;
}

int Torre::operator!() const
{
   int somma = 0;
   for (int i = 0; i < N; i++)
      if (diametri[i] > 0)
         somma++;
   return somma;

}

int Torre::operator%(int n) const
{
   int somma = 0;
   if (n > 0) {
      for (int i = 0; i < N; i++)
         if (diametri[i] == n)
            somma++;
   }
   return somma;
}

istream& operator>>(istream &is, Torre &t)
{
   int n;
   is >> n;
   t += n; // Chiamata implicita al costruttore di conversione
   return is;
}

ostream& operator<<(ostream &os, const Torre &t)
{
   for (int i = 0; i < N; i++)
      if (t.diametri[i] > 0) {
```

```
            for (int j = 0; j < t.diametri[i]; j++)
               os << '*';
            os << ‘\n’;
         }
      return os;
   }
```

Esercizio n. 15 – Pila

Una *pila* è formata un numero limitato di dischi sovrapposti. Il numero dei dischi nella pila è detto *altezza* della pila. Ciascuna pila ha un'altezza massima, pari al numero massimo di dischi che la pila può contenere. Ciascun disco ha un colore, bianco o nero, ed un valore, espresso da un numero intero positivo. In ogni dato istante vale la *regola dei colori*: il numero di dischi bianchi e quello dei dischi neri possono differire di al più una unità (pertanto, ad esempio, se a un dato istante il numero dei dischi bianchi nella pila è pari a 5, il numero dei dischi neri può essere pari a 4, 5 o 6). Le operazioni che possono essere effettuate su una pila sono le seguenti:

- `Pila s;`
  Costruttore di default, che inizializza una pila *s* avente altezza massima pari ad un disco. Inizialmente la pila è vuota.
- `Pila s(n);`
  Costruttore che inizializza una pila *s* avente altezza massima pari a *n* dischi. Inizialmente la pila è vuota.
- `Pila s1(s);`
  Costruttore di copia, che inizializza la pila *s1* col valore della pila *s*.
- `s1 = s`
  Operatore di assegnamento, che sostituisce il valore della pila risultato *s1* con quello della pila *s*.
- `s.inserisci(v, c)`
  Operazione che inserisce nella posizione più in alto della pila *s* un disco di colore *c* e valore *v*. Se l'inserzione porterebbe ad una violazione della regola dei colori, l'inserzione fallisce e la pila resta inalterata. L'operazione ritorna un'indicazione di successo (1) o fallimento (0).
- `s.estrai(v, c)`
  Operazione che estrae dalla pila *s* un disco di colore *c* e valore *v*. Se la pila non contiene almeno un disco di colore *c* e valore *v* oppure l'estrazione porterebbe ad una violazione della regola dei colori, l'estrazione fallisce e la pila resta inalterata. L'operazione ritorna un'indicazione di successo (1) o fallimento (0).
- `s.undo()`
  Operazione che annulla gli effetti dell'operazione di inserzione nella pila *s* o di estrazione dalla pila *s* eseguita più di recente. Nel caso di più esecuzioni consecutive di *undo*(), le esecuzioni successive alla prima lasciano la pila inalterata (cioè, è possibile annullare gli effetti solo dell'operazione di inserzione o estrazione più recente).
- `s % v`
  Operatore di modulo, che ritorna il numero di dischi aventi valore *v* contenuti nella pila *s*.
- `!s`
  Operatore di negazione logica, che ritorna il numero totale di dischi che formano la pila *s*.
- `cout << s`
  Operatore di uscita per il tipo *Pila*. L'uscita ha la forma

  ```
  4:B
  12:N
  1:N
  ```

  In questo esempio, la pila è formata da due dischi neri ed un disco bianco. Il disco bianco è posizionato nella posizione più alta della pila ed ha valore 4.
- `~Pila()`
  Distruttore.

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

Soluzione

```
#include <iostream>
using namespace std;

enum colore { BIANCO = -1, NERO = +1};

class Pila {
   friend ostream& operator<<(ostream&, const Pila&);
   struct disco {
      int val;
      colore col;
   };
   int num_dischi;
   int top;
   disco *vet;
   struct {
      char op;
// operazione da annullare: 'i' per inserzione, 'e' per estrazione,
// '-' per nessuna operazione
      int pos;
      int val;
      colore col;
// Nel caso di estrazione, pos indica la posizione della pila dalla
// quale ho estratto un disco, v indica il valore del disco estratto e
// col indica il colore del disco estratto
   } annulla;
   int violazione(colore, char);
public:
   Pila(int = 1);
   Pila(const Pila&);
   Pila& operator=(const Pila&);
   int inserisci(int, colore);
   int estrai(int, colore);
   void undo();
   int operator%(int) const;
   int operator!() const { return top; }
   ~Pila() { delete[] vet; }
};

int Pila::violazione(colore c, char op)
// Ritorna 1 se l'esecuzione dell'operazione op porterebbe ad una
// violazione della regola dei colori, e 0 altrimenti
{
   int delta = 0;
   for (int i = 0; i < top; i++)
      delta += vet[i].col;
   if (delta == 0)
      return 0;
   if (op == 'i' && ( (delta == BIANCO && c == NERO) ||
        (delta == NERO && c == BIANCO) ) )
      return 0;
   if (op == 'e' && ( (delta == BIANCO && c == BIANCO) ||
        (delta == NERO && c == NERO) ) )
      return 0;
   return 1;
```

```
}

Pila::Pila(int n)
{
   num_dischi = (n > 1) ? n : 1;
   annulla.op = '-';
   top = 0;
   vet = new disco[num_dischi];
}

Pila::Pila(const Pila& p)
{
   num_dischi = p.num_dischi;
   annulla = p.annulla;
   top = p.top;
   vet = new disco[num_dischi];
   for (int i = 0; i < top; i++)
      vet[i] = p.vet[i];
}

Pila& Pila::operator=(const Pila& p)
{
   if (this != &p) {
      annulla = p.annulla;
      top = p.top;
      if (num_dischi != p.num_dischi) {
         delete[] vet;
         num_dischi = p.num_dischi;
         vet = new disco[num_dischi];
      }
      for (int i = 0; i < top; i++)
         vet[i] = p.vet[i];
   }
   return *this;
}

int Pila::inserisci(int v, colore c)
{
   if (top == num_dischi || v <= 0 || violazione(c, 'i'))
      return 0;
   vet[top].val = v;
   vet[top].col = c;
   annulla.op = 'i';
   top++;
   return 1;
}

int Pila::estrai(int v, colore c)
{
   if (top == 0 || v <= 0 || violazione(c, 'e'))
      return 0;
   for (int i = 0; i < top; i++)
      if (vet[i].val == v && vet[i].col == c) {
         annulla.val = v;
         annulla.col = c;
         annulla.pos = i;
         annulla.op = 'e';
         for (int j = i + 1; j < top; j++)
            vet[j - 1] = vet[j];
         top--;
         return 1;
      }
```

```
   return 0;
}

void Pila::undo()
{
   if (annulla.op == 'i')
      top--;
   else if (annulla.op == 'e') {
      for (int i = top; i > annulla.pos; i--)
         vet[i] = vet[i - 1];
      vet[annulla.pos].val = annulla.val;
      vet[annulla.pos].col = annulla.col;
      top++;
   }
   annulla.op = '-';
}

int Pila::operator%(int v) const
{
   int conta = 0;
   for (int i = 0; i < top; i++)
      conta += (vet[i].val == v);
   return conta;
}

ostream& operator<<(ostream& os, const Pila &p)
{
   for (int i = p.top - 1; i >= 0; i--)
      if (p.vet[i].col == BIANCO)
         os << p.vet[i].val << ":B" << '\n';
      else
         os << p.vet[i].val << ":N" << '\n';
   return os;
}
```

Esercizio n. 16 – Matrice binaria

Una *matrice binaria* è formata da elementi aventi valore intero 0 o 1. Il numero di colonne della matrice binaria è pari al numero di bit che formano un *unsigned long int*. Le righe e le colonne sono numerate a partire da 0. Le operazioni che possono essere effettuate su una matrice binaria sono le seguenti:

- `MatriceBinaria s;`
  Costruttore di default, che inizializza una matrice binaria *s* formata da una sola riga. Inizialmente, tutti gli elementi della matrice binaria hanno valore 0.
- `MatriceBinaria s(n);`
  Costruttore che inizializza una matrice binaria *s* formata da *n* righe. Inizialmente, tutti gli elementi della matrice binaria hanno valore 0.
- `MatriceBinaria s1(s);`
  Costruttore di copia, che inizializza la matrice binaria *s1* col valore della matrice binaria *s*.
- `s1 = s`
  Operatore di assegnamento, che sostituisce il valore della matrice binaria risultato *s1* con quello della matrice binaria *s*.
- `s.leggi(b, i, j)`
  Operazione che ritorna il valore *b* (intero 0 o 1) dell'elemento di riga *i* e colonna *j* della matrice binaria *s*. L'operazione ritorna inoltre un'indicazione di successo (1) o fallimento (0).
- `s.scrivi(b, i, j)`
  Operazione che assegna il valore *b* (intero 0 o 1) all'elemento di riga *i* e colonna *j* della matrice binaria *s*. L'operazione ritorna un'indicazione di successo (1) o fallimento (0).
- `s1 & s2`
  Operatore di AND bit a bit, che ritorna una matrice binaria i cui elementi sono il risultato dello AND logico dei corrispondenti elementi delle matrici binarie *s1* ed *s2*. Il numero di righe di *s2* deve essere uguale al numero di righe di *s1* ed è uguale al numero di righe della matrice risultato.
- `s1 &= s2`
  Operatore di AND bit a bit e assegnamento, che assegna a ciascun elemento della matrice binaria *s1* il risultato dello AND logico dei corrispondenti elementi delle matrici binarie *s1* ed *s2*. Il numero di righe di *s2* deve essere uguale al numero di righe di *s1*.
- `s1 | s2`
  Operatore di OR bit a bit, che ritorna una matrice binaria i cui elementi sono il risultato dello OR logico dei corrispondenti elementi delle matrici binarie *s1* ed *s2*. Il numero di righe di *s2* deve essere uguale al numero di righe di *s1* ed è uguale al numero di righe della matrice risultato.
- `s1 |= s2`
  Operatore di OR bit a bit e assegnamento, che assegna a ciascun elemento della matrice binaria *s1* il risultato dello OR logico dei corrispondenti elementi delle matrici binarie *s1* ed *s2*. Il numero di righe di *s2* deve essere uguale al numero di righe di *s1*.
- `cout << s`
  Operatore di uscita per il tipo *MatriceBinaria*. L'uscita ha la forma seguente:

  ```
  0  1  0  …  1
  1  1  1  …  1
  0  0  0  …  0
  ```

  In questo esempio, la matrice binaria *s* è formata da tre righe.
- `~MatriceBinaria()`
  Distruttore.

Mediante il linguaggio C++, realizzare il tipo di dati astratti *MatriceBinaria* definito dalle precedenti specifiche. *Rappresentare ogni riga della matrice binaria mediante un* unsigned long int *interpretato come vettore di bit.* Individuare le eventuali situazioni di errore, e metterne in opera un corretto trattamento.

Soluzione

```
#include <iostream>
using namespace std;

/* Nella soluzione proposta, la colonna 0 della matrice (normalmente
disegnata a sinistra), corrisponde ai bit meno significativi (normalmente
disegnati a destra) degli unsigned long int che rappresentano le righe
della matrice. Dal punto di vista grafico, pertanto, la matrice è
rappresentata in modo speculare. */

class MatriceBinaria {
   friend ostream& operator<<(ostream&, const MatriceBinaria&);
   friend MatriceBinaria
   operator&(const MatriceBinaria&, const MatriceBinaria&);
   friend MatriceBinaria
   operator|(const MatriceBinaria&, const MatriceBinaria&);
   unsigned long int *riga;
   int numRighe;
   static const int numColonne;
public:
   MatriceBinaria(int = 1);
   MatriceBinaria(const MatriceBinaria&);
   MatriceBinaria& operator=(const MatriceBinaria&);
   int leggi(int&, int, int);
   int scrivi(int, int, int);
   MatriceBinaria& operator&=(const MatriceBinaria&);
   MatriceBinaria& operator|=(const MatriceBinaria&);
   ~MatriceBinaria() { delete[] riga; }
};

const int MatriceBinaria::numColonne = 8 * sizeof(unsigned long int);

MatriceBinaria::MatriceBinaria(int n)
{
   numRighe = (n > 0) ? n : 1;
   riga = new unsigned long int[numRighe];
   for (int i = 0; i < numRighe; i++)
      riga[i] = 0;
}

MatriceBinaria::MatriceBinaria(const MatriceBinaria& s)
{
   numRighe = s.numRighe;
   riga = new unsigned long int[numRighe];
   for (int i = 0; i < numRighe; i++)
      riga[i] = s.riga[i];
}

MatriceBinaria& MatriceBinaria::operator=(const MatriceBinaria& s)
{
   if (this != &s) {
      if (numRighe != s.numRighe) {
         delete[] riga;
         numRighe = s.numRighe;
```

```
         riga = new unsigned long int[numRighe];
      }
      for (int i = 0; i < numRighe; i++)
         riga[i] = s.riga[i];
   }
   return *this;
}

int MatriceBinaria::leggi(int& b, int i, int j)
{
   if (i >= 0 && i < numRighe && j >= 0 && j < numColonne) {
      b = ( (riga[i] & (1 << j) ) != 0 );
// != serve a trasformare in 1 qualsiasi valore diverso da 0, necessario
// se j > 0
      return 1;
   }
   return 0;
}

int MatriceBinaria::scrivi(int b, int i, int j)
{
   if ( (i >= 0 && i < numRighe) && (j >= 0 && j < numColonne) &&
         (b == 0 || b == 1)) {
      if (b)
         riga[i] |= (1 << j);
      else
         riga[i] &= ~(1 << j);
      return 1;
   }
   return 0;
}

MatriceBinaria& MatriceBinaria::operator&=(const MatriceBinaria& s)
{
   if (s.numRighe == numRighe)
      for (int i = 0; i < numRighe; i++)
         riga[i] &= s.riga[i];
   return *this;
}

MatriceBinaria& MatriceBinaria::operator|=(const MatriceBinaria& s)
{
   if (s.numRighe == numRighe)
      for (int i = 0; i < numRighe; i++)
         riga[i] |= s.riga[i];
   return *this;
}

MatriceBinaria
operator&(const MatriceBinaria& s1, const MatriceBinaria& s2)
{
   MatriceBinaria m(s1);
   return m &= s2;
}

MatriceBinaria
operator|(const MatriceBinaria& s1, const MatriceBinaria& s2)
{
   MatriceBinaria m(s1);
   return m |= s2;
}
```

```
ostream& operator<<(ostream& os, const MatriceBinaria& s)
{
   for (int i = 0; i < s.numRighe; i++) {
      for (int j = 0; j < s.numColonne; j++) {
         os << ( (s.riga[i] & (1 << j)) != 0 );
         if (j != s.numColonne - 1)
            os << '\t';
      }
      os << ‘\n’;
   }
   return os;
}
```

Esercizio n. 17 – Molteplicità

Una *molteplicità* è formata da elementi aventi valore intero compreso tra 0 ed il valore di una costante globale *MAX*, estremi inclusi. Il numero massimo di elementi che possono avere lo stesso valore è chiamato la *valenza* della molteplicità. Le operazioni che possono essere effettuate su una molteplicità sono le seguenti:

- `Molteplicità s;`
  Costruttore di default, che inizializza una molteplicità *s* avente valenza 1. Inizialmente, la molteplicità è vuota.
- `Molteplicità s(n);`
  Costruttore che inizializza una molteplicità *s* avente valenza *n*. Inizialmente, la molteplicità è vuota.
- `Molteplicità s1(s);`
  Costruttore di copia, che inizializza la molteplicità *s1* col valore della molteplicità *s*.
- `s1 = s`
  Operatore di assegnamento, che sostituisce il valore della molteplicità risultato *s1* con quello della molteplicità *s*.
- `s += v`
  Operatore di somma e assegnamento, che inserisce nella molteplicità *s* un elemento avente valore *v*.
- `s -= v`
  Operatore di sottrazione e assegnamento, che estrae dalla molteplicità *s* un elemento avente valore *v*.
- `s.annulla(n)`
  Operazione che annulla le *n* operazioni di somma e assegnamento e/o sottrazione e assegnamento che sono state effettuate più di recente nella molteplicità *s*. L'esecuzione anche ripetuta di questa operazione è in grado di annullare un numero complessivo di operazioni non superiore al valore di una costante globale *ANNULLA_MAX*.
- `s % v`
  Operatore di modulo, che ritorna il numero di elementi della molteplicità *s* che hanno valore *v.*
- `!s`
  Operatore di negazione logica, che ritorna il numero totale di elementi che formano la molteplicità *s*.
- `cout << s`
  Operatore di uscita per il tipo *Molteplicità*. L'uscita ha la forma seguente:

  ```
  [5] <0:3, 1:2, 10:4>
  ```

  In questo esempio, la molteplicità *s* ha valenza 5. Tre elementi hanno valore 0, due elementi hanno valore 1 e quattro hanno valore 10.
- `~Molteplicità()`
  Distruttore.

Mediante il linguaggio C++, realizzare il tipo di dati astratti *Molteplicità* definito dalle precedenti specifiche, nell'ipotesi di valori ragionevolmente piccoli delle costanti globali *MAX* e *ANNULLA_MAX*. Individuare le eventuali situazioni di errore, e metterne in opera un corretto trattamento.

Soluzione

```
#include <iostream>
using namespace std;
```

```
const int MAX = 20; // Ad esempio
const int ANNULLA_MAX = 10; // Ad esempio

class Molteplicita {
   friend ostream& operator<<(ostream&, const Molteplicita&);

   class PilaCircolare {
   public:
      enum codice { ADD, SUB };
      PilaCircolare() { num = top = 0; }
      void inserisci(codice, int);
      int estrai(codice&, int&);
   private:
      struct elem {
         codice op;
         int val;
      };
      elem buf[ANNULLA_MAX];
      int top; // indice del primo elemento libero in buf
      int num; // numero di elementi occupati in buf
   };

   PilaCircolare annullaBuffer;
   int valenza;
   int elemento[MAX + 1];
// MAX è ragionevolmente piccolo, e pertanto è giustificata una
// realizzazione a vettore
public:
   Molteplicita(int = 1);
// Molteplicita(const Molteplicita&);
// Molteplicita& operator=(const Molteplicita s&);
   Molteplicita& operator+=(int);
   Molteplicita& operator-=(int);
   Molteplicita& annulla(int);
   int operator%(int) const;
   int operator!() const;
// ~Molteplicita();
};

/* ------------- Classe PilaCircolare ------------ */

void Molteplicita::PilaCircolare::inserisci(codice c, int v)
{
// Se buf è pieno, scrivo sopra l'elemento inserito in buf da più tempo.
   if (num < ANNULLA_MAX)
      num++;
   buf[top].op = c;
   buf[top].val = v;
   top = (top == ANNULLA_MAX - 1) ? 0 : top + 1; // Buffer circolare
}

int Molteplicita::PilaCircolare::estrai(codice& c, int& v)
{
   if (num > 0) {
      num--;
      top = (top == 0) ? ANNULLA_MAX - 1 : top - 1; // Buffer circolare
      v = buf[top].val;
      c = buf[top].op;
      return 1;
   }
   return 0;
```

```
}

/* ------------- Classe Molteplicita ------------- */

Molteplicita::Molteplicita(int n)
{
   valenza = (n > 0) ? n : 1;
   for (int i = 0; i <= MAX; i++)
      elemento[i] = 0;
}

Molteplicita& Molteplicita::operator+=(int v)
{
   if (v >= 0 && v <= MAX && elemento[v] < valenza) {
      elemento[v]++;
      annullaBuffer.inserisci(PilaCircolare::ADD, v);
   }
   return *this;
}

Molteplicita& Molteplicita::operator-=(int v)
{
   if (v >= 0 && v <= MAX && elemento[v] > 0) {
      elemento[v]--;
      annullaBuffer.inserisci(PilaCircolare::SUB, v);
   }
   return *this;
}

Molteplicita& Molteplicita::annulla(int n)
{
   If (n > 0) {
      PilaCircolare::codice cod;
      int v;
      for (int i = 0; i < n && annullaBuffer.estrai(cod, v); i++)
         cod == PilaCircolare::ADD ? elemento[v]-- : elemento[v]++;
   }
   return *this;
}

int Molteplicita::operator%(int v) const
{
   if (v >= 0 && v <= MAX)
      return elemento[v];
   return -1;
}

int Molteplicita::operator!() const
{
   int conta = 0;
   for (int i = 0; i <= MAX; i++)
      conta += elemento[i];
   return conta;
}

ostream& operator<<(ostream &os, const Molteplicita& m)
{
   os << '[' << m.valenza << "] <";
   int primo = 1;
   for (int i = 0; i <= MAX; i++)
      if (m.elemento[i] >= 0) {
         if (primo)
```

```
            primo = 0;
         else
            os << ", ";
         os << i << ':' << m.elemento[i];
      }
   os << '>';
   return os;
}
```

Esercizio n. 18 – Pannello

Un *pannello* è formato da caselle numerate a partire da 0. Ciascuna casella può essere libera oppure occupata da uno o più cubi sovrapposti. Ciascun cubo ha un valore intero maggiore o uguale ad 1. In una data casella, i cubi sono sovrapposti in ordine di valore decrescente. Il numero massimo dei cubi sovrapponibili su una casella è dato dal valore della costante globale *CMAX*. Un cubo è *inseribile* in una data casella se la casella è libera oppure se il valore di quel cubo è pari o inferiore al valore del cubo che occupa la posizione più in alto tra i cubi già sovrapposti su quella casella. Un cubo è *estraibile* da una data casella se occupa la posizione più in alto tra i cubi sovrapposti su quella casella. Il *peso* di una data casella è dato dalla somma dei valori dei cubi sovrapposti su quella casella. Le operazioni che possono essere effettuate su un pannello sono le seguenti:

- `Pannello s;`
  Costruttore di default, che inizializza un pannello *s* formato da una sola casella. Inizialmente, tale casella è vuota.
- `Pannello s(n);`
  Costruttore che inizializza un pannello *s* formato da *n* caselle. Inizialmente, tutte le caselle del pannello sono vuote.
- `Pannello s1(s);`
  Costruttore di copia, che inizializza il pannello *s1* col valore del pannello *s*.
- `s1 = s`
  Operatore di assegnamento, che sostituisce il valore del pannello risultato *s1* con quello del pannello *s*.
- `s += v`
  Operatore di somma e assegnamento, che inserisce nel pannello *s* un cubo avente valore *v*. Il cubo viene inserito nella casella avente il più basso numero d'ordine tra le caselle in cui quel cubo è inseribile.
- `s -= v`
  Operatore di sottrazione e assegnamento, che estrae dal pannello *s* un cubo avente valore *v*. Il cubo viene estratto dalla casella a più basso numero d'ordine che contiene un cubo estraibile avente valore *v*.
- `s.confronta(i, j)`
  Operazione che ritorna il valore -1 se, nel pannello *s*, il peso della casella *i* è minore del peso della casella *j*; ritorna il valore 0 se le due caselle hanno lo stesso peso; e ritorna il valore +1 se il peso della casella *i* è maggiore del peso della casella *j*.
- `s % i`
  Operatore di modulo, che ritorna il peso della casella *i* del pannello *s.*
- `!s`
  Operatore di negazione, che ritorna la somma dei pesi delle caselle che formano il pannello *s*.
- `cout << s`
  Operatore di uscita per il tipo *Pannello*. L'uscita consiste nei pesi delle caselle che formano il pannello racchiusi tra parentesi angolate e separati da virgole, come nell'esempio seguente:

  ```
  <44, 21, 1, 0>
  ```

  In questo esempio, il pannello *s* è formato da 4 caselle. La prima casella ha peso 44, l'ultima è libera.
- `~Pannello()`
  Distruttore.

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

Soluzione

```
#include <iostream>
using namespace std;

const int CMAX = 10; // Ad esempio

class Pannello {

   class Casella {
      int cubo[CMAX];
      int testa; // indice del primo elemento libero
      int inseribile(int) const;
      int estraibile(int) const;
   public:
      Casella();
      int operator!() const;
      int operator+=(int);
      int operator-=(int);
   };

   friend ostream& operator<<(ostream&, const Pannello&);
   int dim; // Numero di caselle che formano il pannello
   Casella* vet;
public:
   Pannello(int = 1);
   Pannello(const Pannello&);
   Pannello& operator=(const Pannello&);
   Pannello& operator+=(int);
   Pannello& operator-=(int);
   int confronta(int, int) const;
   int operator%(int) const;
   int operator!() const;
   ~Pannello() { delete[] vet; }
};

/* ------------- Classe Casella ------------ */

inline Pannello::Casella::Casella()
{
   testa = 0;
}

inline int Pannello::Casella::inseribile(int v) const
{
   return (v >= 1) && (testa <= CMAX - 1) &&
        ( (testa == 0) || (cubo[testa - 1] >= v) );
}

inline int Pannello::Casella::estraibile(int v) const
{
   return (testa > 0) && (cubo[testa - 1] == v);
}

int Pannello::Casella::operator!() const
{
```

```
   int somma = 0;
   for (int i = 0; i < testa; i++)
      somma += cubo[i];
   return somma;
}

int Pannello::Casella::operator+=(int v)
// Ritorna 1 se l'inserzione ha successo, e 0 altrimenti
{
   if (inseribile(v)) {
      cubo[testa++] = v;
      return 1;
   }
   return 0;
}

int Pannello::Casella::operator-=(int v)
// Ritorna 1 se l'estrazione ha successo, e 0 altrimenti
{
   if (estraibile(v)) {
      --testa;
      return 1;
   }
   return 0;
}

/* ------------- Classe Pannello ----------- */

Pannello::Pannello(int n)
{
   dim = (n > 0) ? n : 1;
   vet = new Casella[dim];
}

Pannello::Pannello(const Pannello& s)
{
   dim = s.dim;
   vet = new Casella[dim];
   for (int i = 0; i < dim; i++)
      vet[i] = s.vet[i];
}

Pannello& Pannello::operator=(const Pannello& s)
{
   if (this != & s) {
      if (dim != s.dim) {
         delete[] vet;
         dim = s.dim;
         vet = new Casella[dim];
      }
      for (int i = 0; i < dim; i++)
         vet[i] = s.vet[i];
   }
   return *this;
}

Pannello& Pannello::operator+=(int v)
// Se l'inserzione non può avere luogo, l'operazione lascia
// il pannello inalterato
{
   for (int i = 0; i < dim && !(vet[i] += v); i++);
   return *this;
```

```
}

Pannello& Pannello::operator-=(int v)
// Se l'estrazione non può avere luogo, l'operazione lascia
// il pannello inalterato
{
   for (int i = 0; i < dim && !(vet[i] -= v); i++);
   return *this;
}

int Pannello::confronta(int i, int j) const
{
   i = (i < 0) ? 0 : (i >= dim) ? dim - 1 : i;
   j = (j < 0) ? 0 : (j >= dim) ? dim - 1 : j;
   return !vet[i] < !vet[j] ? -1 : ( !vet[i] == !vet[j] ? 0 : 1 )
}

int Pannello::operator%(int i) const
{
   i = (i < 0) ? 0 : (i >= dim) ? dim - 1 : i;
   return !vet[i];
}

int Pannello::operator!() const
{
   int somma = 0;
   for (int i = 0; i < dim; i++)
      somma += !vet[i];
   return somma;
}

ostream& operator<<(ostream& os, const Pannello& s)
{
   os << '<';
   for (int i = 0; i < s.dim; i++) {
      os << !s.vet[i];
      if (i != s.dim - 1)
         os << ", ";
   }
   os << '>';
   return os;
}
```

Esercizio n. 19 – Piramide

Una *piramide* è formata da elementi aventi valore intero, organizzati in un numero limitato di righe. Le righe sono numerate a partire da 0. La prima riga della piramide ha un solo elemento, la seconda riga ha due elementi, la terza riga ha tre elementi e così via. Gli elementi di una data riga sono numerati a partire da 0. Le operazioni che possono essere effettuate su una piramide sono le seguenti:

- `Piramide s;`
  Costruttore di default, che inizializza una piramide *s* formata da una sola riga, il cui elemento ha valore iniziale 0.
- `Piramide s(n);`
  Costruttore che inizializza una piramide *s* formata da *n* righe. Tutti gli elementi della piramide hanno valore iniziale 0.
- `Piramide s1(s);`
  Costruttore di copia, che inizializza la piramide *s1* col valore della piramide *s*.
- `s1 = s`
  Operatore di assegnamento, che sostituisce il valore della piramide risultato *s1* con quello della piramide *s*.
- `s.nuovoValore(m, n, v)`
  Operazione che assegna il valore *v* all'elemento *n* della riga *m* della piramide *s*.
- `s.valore(m, n, v)`
  Operazione che ritorna il valore *v* dell'elemento *n* della riga *m* della piramide *s*. L'operazione ritorna inoltre un'indicazione di successo (1) o fallimento (0).
- `s.specchia()`
  Operazione che inverte l'ordine dei valori degli elementi di ciascuna riga della piramide *s* (e pertanto, ad esempio, in una data riga il primo elemento assume il valore dell'ultimo elemento, e il secondo elemento assume il valore del penultimo elemento).
- `cout << s`
  Operatore di uscita per il tipo *Piramide*. L'uscita ha la forma

```
 9
13 -15
 0  -2   3
 9   7  -6 -18
```

  In quest'esempio, la piramide *s* è formata da quattro righe.
- `~Piramide()`
  Distruttore.

Mediante il linguaggio C++, realizzare il tipo di dati astratti *Piramide* definito dalle precedenti specifiche, in modo tale che sia possibile concatenare le operazioni *nuovoValore*() e *specchia*(), come negli esempi seguenti:

```
s.nuovoValore(m, n, v).specchia();
s.specchia().nuovoValore(m, n, v);
```

Individuare le eventuali situazioni di errore, e metterne in opera un corretto trattamento.

Soluzione

```
#include <iostream>
using namespace std;

class Piramide {
   friend ostream operator<<(ostream& os, const Piramide& s);
   int dim;
```

```
   int *array;
   int inizio_riga(int m) { return m * (m + 1) / 2; }
// Ritorna l'indice dell'elemento di array corrispondente al primo
// elemento della riga m
public:
   Piramide(int = 1);
   Piramide(const Piramide&);
   Piramide& operator=(const Piramide&);
   Piramide& nuovoValore(int, int, int);
   int valore(int, int, int&) const;
   Piramide& specchia();
   ~Piramide() { delete[] array; }
};

Piramide::Piramide(int n)
{
   dim = (n > 0) ? n : 1;
   array = new int[dim * (dim + 1) / 2];
   for (int i = 0; i < dim * (dim + 1) / 2; i++)
      array[i] = 0;
}

Piramide::Piramide(const Piramide& s)
{
   dim = s.dim;
   array = new int[dim * (dim + 1) / 2];
   for (int i = 0; i < dim * (dim + 1) / 2; i++)
      array[i] = s.array[i];
}

Piramide& Piramide::operator=(const Piramide& s)
{
   if (this != &s) {
      delete[] array;
      dim = s.dim;
      array = new int[dim * (dim + 1) / 2];
      for (int i = 0; i < dim * (dim + 1) / 2; i++)
         array[i] = s.array[i];
   }
   return *this;
}

Piramide& Piramide::nuovoValore(int m, int n, int v)
{
   if (m >= 0 && m < dim)     // le righe sono numerate a partire da 0
      if (n >= 0 && n <= m)   // la riga m ha m + 1 elementi
         array[inizio_riga(m) + n] = v;
   return *this;
}

int Piramide::valore(int m, int n, int &v) const
{
   if (m >= 0 && m < dim)
      if (n >= 0 && n <= m) {
         v = array[inizio_riga(m) + n];
         return 1;
      }
   return 0;
}

Piramide& Piramide::specchia()
{
```

```
   for (int i = 1; i < dim; i++) {
      int pos = inizio_riga(i);
      for (int j = 0; j < (i + 1) / 2; j++) {
         int tmp = array[pos + j];
         array[pos + j] = array[pos + i - j];
         array[pos + i - j] = tmp;
      }
   }
   return *this;
}

ostream& operator<<(ostream& os, const Piramide& s)
{
   int v;
   for (int i = 0; i < s.dim; i++) {
      for (int j = 0; j <= i; j++) {
         s.valore(i, j, v);
         os << v;
         if (j != i)
            os << '\t';
      }
      os << '\n';
   }
   return os;
}
```

Esercizio n. 20 – Gruppo

Una *gruppo* è in grado di contenere un numero illimitato di elementi aventi valore intero positivo o nullo. Il massimo valore degli elementi del gruppo è pari al numero di bit che formano un *unsigned int* diminuito di 1 (e pertanto, ad esempio, se un *unsigned int* è rappresentato su 32 bit, i valori degli elementi saranno compresi tra 0 e 31, estremi inclusi). Le operazioni che possono essere effettuate su un gruppo sono specificate di seguito. In alcune operazioni, un selettore *sel*, di tipo *unsigned int*, viene interpretato come vettore di bit e specifica valori di elementi. I bit del selettore sono numerati a partire dal meno significativo (il bit 0). Se il bit *i*-esimo del selettore vale 1, allora il selettore include il valore *i*.

- `Gruppo s;`
  Costruttore di default, che inizializza un gruppo *s*. Inizialmente il gruppo è vuoto.
- `Gruppo s1(s);`
  Costruttore di copia, che inizializza il gruppo *s1* col valore del gruppo *s*.
- `s1 = s`
  Operatore di assegnamento, che sostituisce il valore del gruppo risultato *s1* con quello del gruppo *s*.
- `s += v`
  Operatore di somma e assegnamento che inserisce nel gruppo *s* un nuovo elemento avente valore *v*.
- `s += sel`
  Operatore di somma e assegnamento che inserisce nel gruppo *s* un nuovo elemento per ciascuno dei valori inclusi nel selettore *sel*.
- `s -= v`
  Operatore di sottrazione e assegnamento che estrae dal gruppo *s* un elemento avente valore *v*.
- `s -= sel`
  Operatore di sottrazione e assegnamento che estrae dal gruppo *s* un elemento per ciascuno dei valori inclusi nel selettore *sel*.
- `s ^= v`
  Operatore di XOR bit a bit e assegnamento che estrae dal gruppo *s* tutti gli elementi aventi valore *v*.
- `s ^= sel`
  Operatore di XOR bit a bit e assegnamento che estrae dal gruppo *s* tutti gli elementi il cui valore è incluso nel selettore *sel*.
- `s % v`
  Operatore di modulo che ritorna il numero di elementi del gruppo *s* che hanno valore *v*.
- `s % sel`
  Operatore di modulo che ritorna il numero totale degli elementi del gruppo *s* il cui valore è incluso nel selettore *sel*.
- `cout << s`
  Operatore di uscita per il tipo *Gruppo*. L'uscita ha la forma seguente:

  ```
  [18] <0:5, 3:1, 4:12>
  ```

  In quest'esempio, il gruppo contiene 18 elementi in tutto. Cinque elementi hanno valore 0, un elemento ha valore 3 e 12 hanno valore 4.
- `~Gruppo()`
  Distruttore.

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

Soluzione

```
#include <iostream>
using namespace std;

const int NUM = 8 * sizeof(unsigned int);

class Gruppo
{
   friend ostream& operator<<(ostream&, const Gruppo&);
   int elem[NUM];
public:
   Gruppo();
// Gruppo(const Gruppo&);
// Gruppo& operator=(const Gruppo&);
   Gruppo& operator+=(int);
   Gruppo& operator+=(unsigned int);
   Gruppo& operator-=(int);
   Gruppo& operator-=(unsigned int);
   Gruppo& operator^=(int);
   Gruppo& operator^=(unsigned int);
   int operator%(int) const;
   int operator%(unsigned int) const;
// ~Gruppo();
};

Gruppo::Gruppo()
{
   for (int i = 0; i < NUM; i++)
      elem[i] = 0;
}

Gruppo& Gruppo::operator+=(int v)
{
   if (v >= 0 && v < NUM)
      elem[v]++;
   return *this;
}

Gruppo& Gruppo::operator+=(unsigned int sel)
{
   for (int i = 0; i < NUM; i++)
      if (sel & 1 << i)
         elem[i]++;
   return *this;
}

Gruppo& Gruppo::operator-=(int v)
{
   if (v >= 0 && v < NUM && elem[v] > 0)
      elem[v]--;
   return *this;

}

Gruppo& Gruppo::operator-=(unsigned int sel)
{
   for (int i = 0; i < NUM; i++)
      if ((sel & 1 << i) && (elem[i] > 0))
         elem[i]--;
   return *this;
}
```

```
Gruppo& Gruppo::operator^=(int v)
{
   if (v >= 0 && v < NUM)
      elem[v] = 0;
   return *this;
}

Gruppo& Gruppo::operator^=(unsigned int sel)
{
   for (int i = 0; i < NUM; i++)
      if (sel & 1 << i)
         elem[i] = 0;
   return *this;
}

int Gruppo::operator%(int v) const
{
   if (v >= 0 && v < NUM)
      return elem[v];
   return 0;
}

int Gruppo::operator%(unsigned int sel) const
{
   int somma = 0;
   for (int i = 0; i < NUM; i++)
      if (sel & 1 << i)
         somma += elem[i];
   return somma;
}

ostream& operator<<(ostream& os, const Gruppo& g)
{
   os << '[' << g % ~(0u) << "] <";
   int primo = 1;
   for (int i = 0; i < NUM; i++)
      if (g.elem[i] > 0)
         if (primo) {
            os << i << ':' << g.elem[i];
            primo = 0;
         } else
            os << ", " << i << ':' << g.elem[i];
      return os << '>';
}
```

Esercizio n. 21 – Lista ordinata

Una *lista ordinata* è formata da elementi aventi valore intero strettamente positivo. Gli elementi sono numerati a partire da 1 e sono ordinati secondo valori dispari crescenti seguiti da valori pari decrescenti. Le operazioni che possono essere effettuate su una lista ordinata sono le seguenti:

- `ListaOrdinata s;`
  Costruttore di default, che inizializza una lista ordinata *s*. Inizialmente la lista è vuota.
- `ListaOrdinata s1(s);`
  Costruttore di copia, che inizializza la lista ordinata *s1* col valore della lista ordinata *s*.
- `s1 = s`
  Operatore di assegnamento, che sostituisce il valore della lista ordinata risultato *s1* con quello della lista ordinata *s*.
- `s.inserisci(v)`
  Operazione che inserisce un elemento avente valore *v* nella lista ordinata *s*.
- `s.estrai(v)`
  Operazione che estrae un elemento avente valore *v* dalla lista ordinata *s*. L'operazione ritorna un'indicazione di successo (1) o fallimento (0).
- `s.estraiPari()`
  Operazione che estrae l'elemento avente *il più alto valore pari* e ne ritorna il valore. Se la lista ordinata non contiene elementi a valore pari, l'operazione ritorna 0.
- `s.estraiDispari()`
  Operazione che estrae l'elemento avente *il più basso valore dispari* e ne ritorna il valore. Se la lista ordinata non contiene elementi a valore dispari, l'operazione ritorna 0.
- `s[n]`
  Operatore di indicizzazione, che ritorna il valore dell'*n*-esimo elemento della lista ordinata *s*.
- `!s`
  Operatore di negazione che ritorna il numero totale di elementi che formano la lista ordinata *s*.
- `~ListaOrdinata()`
  Distruttore.

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

Soluzione

```
#include <iostream>
using namespace std;

class ListaOrdinata {
   class Lista {    // ordinamento secondo valori crescenti
      struct elem {
         int val;
         elem *next;
      };
      elem *testa;
      int num;
      void copia(const Lista&);
      void elimina();
   public:
```

```
      Lista() : testa(NULL), num(0) {}
      Lista(const Lista& s) { copia(s); }
      Lista& operator=(const Lista&);
      void inserisci(int);
      int estraiPrimo();      // estrazione in testa
      int estrai(int);        // estrazione per valore
      int operator[](int) const;
      int operator!() const { return num; }
      ~Lista() { elimina(); }
   };
   Lista pari, dispari;
public:
   ListaOrdinata() {}
// ListaOrdinata(const ListaOrdinata&);
// ListaOrdinata& operator=(const ListaOrdinata&);
   void inserisci(int);
   int estrai(int);
   int operator[](int) const;
   int estraiPari() { return -pari.estraiPrimo(); }
   int estraiDispari() { return dispari.estraiPrimo(); }
   int operator!() const { return !pari + !dispari; }
// ~ListaOrdinata();
};

void ListaOrdinata::Lista::copia(const Lista& s)
{
   num = s.num;
   testa = NULL;
   if (s.testa != NULL) {
      testa = new elem;
      testa->val = s.testa->val;
      elem *q = s.testa->next, *t = testa;
      while (q != NULL) {
         t->next = new elem;
         t = t->next;
         t->val = q->val;
         q = q->next;
      }
      t->next = NULL;
   }
}

void ListaOrdinata::Lista::elimina()
{
   elem* aux = testa;
   while (aux != NULL) {
      testa = testa->next;
      delete aux;
      aux = testa;
   }
}

ListaOrdinata::Lista& ListaOrdinata::Lista::operator=(const Lista& s)
{
   if (this != &s) {
      elimina();
      copia(s);
   }
   return *this;
}

void ListaOrdinata::Lista::inserisci(int v)
```

```
{
   elem *prima = testa, *dopo;
   for (dopo = testa; dopo != NULL && dopo->val < v; dopo = dopo->next)
      prima = dopo;
// inserisco un nuovo elemento tra gli elementi puntati da prima e dopo
   elem *tmp = new elem;
   tmp->val = v;
   tmp->next = dopo;
   if (dopo == testa)
      testa = tmp;
   else
      prima->next = tmp;
   num++;
}

int ListaOrdinata::Lista::estraiPrimo()
{
   if (num == 0)
      return 0;
   elem *tmp = testa;
   testa = testa->next;
   int v = tmp->val;
   delete tmp;
   num--;
   return v;
}

int ListaOrdinata::Lista::estrai(int v)
{
   if (testa == NULL)
      return 0;
   elem *tmp, *prec;
   for (tmp = testa; tmp != NULL && tmp->val != v; tmp = tmp->next)
      prec = tmp;
// ora tmp punta l'elemento avente valore v,
// e prec punta l'elemento precedente
   if (tmp == NULL)
      return 0;
   if (tmp == testa)
      testa = tmp->next;
   else
      prec->next = tmp->next;
   delete tmp;
   num--;
   return 1;
}

int ListaOrdinata::Lista::operator[](int n) const
{
   elem *aux = testa;
   for (int i = 0; i < n; i++)
      aux = aux->next;
   return aux->val;
}

void ListaOrdinata::inserisci(int v)
{
   if (v > 0)
      if (v % 2)
         dispari.inserisci(v);
      else
         pari.inserisci(-v);
```

```
// Nella lista pari gli elementi sono memorizzati col segno negativo,
// in modo da poter utilizzare un ordinamento secondo valori crescenti.
}

int ListaOrdinata::estrai(int v)
{
   if (v > 0)
      if (v % 2)
         return dispari.estrai(v);
      else
         return pari.estrai(-v);
   return 0;
}

int ListaOrdinata::operator[](int n) const
{
   if (n <= 0)
      return 0;
   if (n <= !dispari)
      return dispari[n - 1];
   else if (n <= !dispari + !pari)
      return -(pari[n - !dispari - 1]);
   return 0;
}
```

Esercizio n. 22 – Accumulatore

Un *accumulatore* è in grado di contenere un numero illimitato di elementi aventi valore intero. Le operazioni che possono essere effettuate su un accumulatore sono le seguenti:

- `Accumulatore t;`
  Costruttore di default, che inizializza un accumulatore *t*. Inizialmente, l'accumulatore è vuoto.
- `Accumulatore t1(t);`
  Costruttore di copia, che inizializza l'accumulatore *t1* col valore dell'accumulatore *t*.
- `t1 = t`
  Operatore di assegnamento, che sostituisce il valore dell'accumulatore risultato *t1* con quello dell'accumulatore *t*.
- `t.inserisci(v)`
  Operazione che inserisce un elemento avente valore *v* nell'accumulatore *t*.
- `t.estrai(modo, v)`
  Operazione che estrae un elemento dall'accumulatore *t*, e ritorna il valore *v* di tale elemento. Se l'argomento *modo* ha valore negativo, l'elemento estratto è quello avente valore negativo e inserito nell'accumulatore da più tempo; se *modo* vale 0, l'elemento estratto ha valore 0; infine, se *modo* ha valore strettamente positivo, l'elemento estratto è quello avente valore strettamente positivo e inserito nell'accumulatore da più tempo. L'operazione fallisce se l'accumulatore non contiene un elemento con le caratteristiche specificate da *modo*. L'operazione ritorna un'indicazione di successo (1) o fallimento (0).
- `t % modo`
  Operatore di modulo, che ritorna il numero di elementi contenuti nell'accumulatore *t*, e aventi valore negativo se l'argomento *modo* ha valore negativo; aventi valore 0 se *modo* vale 0; e infine, aventi valore strettamente positivo se *modo* ha valore strettamente positivo.
- `!t`
  Operatore di negazione logica, che ritorna il numero totale di elementi contenuti nell'accumulatore *t*.
- `~Accumulatore()`
  Distruttore.

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

Soluzione

```
#include <iostream>
using namespace std;

class Accumulatore {
   class Coda {
      struct elem {
         int val;
         elem *succ;
      };
      elem *primo, *ultimo;
      void copia(const Coda&);
      void elimina();
   public:
      Coda() { primo = ultimo = NULL; }
      Coda(const Coda& s) { copia(s); }
      Coda& operator=(const Coda&);
```

```
      Coda& inserisci(int);
      int estrai(int&);
      int vuota() cost { return primo == NULL; }
      int operator!() const;
      ~Coda() { elimina(); }
   };
   Coda negativi, positivi;
   int zeri;
public:
   Accumulatore() { zeri = 0; }
// Accumulatore(const Accumulatore& s);
// Accumulatore& operator=(const Accumulatore&);
   void inserisci(int);
   int estrai(int, int&);
   int operator%(int);
   int operator!() const {
      return zeri + !positivi + !negativi; }
// ~Accumulatore();
};

// -------- CODA --------

void Accumulatore::Coda::copia(const Coda& s)
{
   primo = ultimo = NULL;
   if (s.primo != NULL) {
      primo = new elem;
      primo->val = s.primo->val;
      if (s.ultimo == s.primo)
         ultimo = primo;
      elem *aux = s.primo->succ, *ultimo = primo;
      while (aux != NULL) {
         ultimo->succ = new elem;
         ultimo = ultimo->succ;
         ultimo->val = aux->val;
         aux = aux->succ;
      }
      ultimo->succ = NULL;
   }
}

void Accumulatore::Coda::elimina()
{
   ultimo = primo;
   while (primo != NULL) {
      primo = primo->succ;
      delete ultimo;
      ultimo = primo;
   }
}

Accumulatore::Coda& Accumulatore::Coda::operator=(const Coda& s)
{
   if (this != &s) {
      elimina();
      copia(s);
   }
   return *this;
}

Accumulatore::Coda& Accumulatore::Coda::inserisci(int d)
{ //Inserzione in coda
```

```
   if (primo == NULL) {
      primo = new elem;
      ultimo = primo;
      primo->val = d;
      primo->succ = NULL;
   }
   else {
      ultimo->succ = new elem;
      ultimo = ultimo->succ;
      ultimo->val = d;
      ultimo->succ = NULL;
   }
   return *this;
}

int Accumulatore::Coda::estrai(int& v)
{ // Estrazione in testa
   if (primo != NULL) {
      elem *tmp = primo;
      primo = primo->succ;
      v = tmp->val;
      delete tmp;
      return 1;
   }
   return 0;
}

int Accumulatore::Coda::operator!() const
{
   int conta = 0;
   elem* aux = primo;
   while (aux != NULL) {
      aux = aux->succ;
      conta++;
   }
   return conta;
}

// -------- ACCUMULATORE --------

void Accumulatore::inserisci(int v)
{
   if (v == 0)
      zeri++;
   else if (v > 0)
      positivi.inserisci(v);
   else
      negativi.inserisci(v);
}

int Accumulatore::estrai(int modo, int& v)
{
   if (modo == 0 && zeri > 0) {
      v = 0;
      zeri--;
      return 1;
   }
   else if (modo > 0 && !positivi.vuota()) {
      positivi.estrai(v);
      return 1;
   }
   else if (modo < 0 && !negativi.vuota()) {
```

```
      negativi.estrai(v);
      return 1;
   }
   return 0;
}

int Accumulatore::operator%(int modo)
{
   if (modo == 0)
      return zeri;
   if (modo > 0)
      return !positivi;
   return !negativi;
}
```

Esercizio n. 23 – Buffer 2

Un *buffer* è in grado di contenere un numero limitato di elementi aventi valore intero e gestiti secondo la strategia *first-in first-out*. Il numero di elementi che il buffer è in grado di contenere è chiamato la *capacità* del buffer. Le operazioni che possono essere effettuate su un buffer sono le seguenti:

- `Buffer t;`
  Costruttore di default, che inizializza un buffer *t*. Inizialmente, il buffer ha capacità unitaria, ed è vuoto.
- `Buffer t(c);`
  Costruttore che inizializza un buffer *t*. Inizialmente, il buffer ha capacità pari a *c*, ed è vuoto.
- `Buffer t1(t);`
  Costruttore di copia, che inizializza il buffer *t1* col valore del buffer *t*.
- `t1 = t`
  Operatore di assegnamento, che sostituisce il valore del buffer risultato *t1* con quello del buffer *t*.
- `t.inserisci(v)`
  Operazione che inserisce un elemento avente valore *v* nel buffer *t*. L'operazione fallisce se il buffer è pieno. L'operazione ritorna un'indicazione di successo (1) o fallimento (0).
- `t.estrai(v)`
  Operazione che estrae un elemento dal buffer *t*, e ritorna il valore *v* di tale elemento. L'operazione fallisce se il buffer è vuoto. L'operazione ritorna un'indicazione di successo (1) o fallimento (0).
- `t.capacita(c)`
  Operazione che modifica la capacità del buffer *t*. La nuova capacità è specificata dal valore dell'argomento *c*. L'operazione fallisce se il buffer contiene un numero di elementi maggiore di *c*. L'operazione ritorna un'indicazione di successo (1) o fallimento (0).
- `t % v`
  Operatore di modulo, che ritorna il numero di elementi aventi valore *v* contenuti nel buffer *t*.
- `!t`
  Operatore di negazione logica, che ritorna il numero di elementi contenuti nel buffer *t*.
- `cout << t`
  Operatore di uscita per il tipo *Buffer*. L'uscita ha la forma seguente:

  ```
  [10] <-4, 0, 4, -3, 55>
  ```

  In questo esempio, il buffer *t* ha capacità pari a 10 elementi e attualmente contiene 5 elementi. L'elemento inserito nel buffer da più tempo ha valore -4.
- `~Buffer()`
  Distruttore.

Mediante il linguaggio C++, realizzare il tipo di dati astratti *Buffer* definito dalle precedenti specifiche. *Fare ricorso ad una rappresentazione del buffer a lista.* Individuare le eventuali situazioni di errore, e metterne in opera un corretto trattamento.

Soluzione

```
#include <iostream>
using namespace std;

class Buffer {
   friend ostream& operator<<(ostream&, const Buffer&);
```

```
   struct elem {
      int val;
      elem *next;
   };
   elem *testa, *coda;
   int num, cap;
   void copia(const Buffer&);
   void elimina();
public:
   Buffer(int = 1);
   Buffer(const Buffer& s) { copia(s); }
   Buffer& operator=(const Buffer&);
   int inserisci(int);
   int estrai(int&);
   int capacita(int);
   int operator!() const { return num; }
   int operator%(int) const;
   ~Buffer() { elimina(); }
};

Buffer::Buffer(int c)
{
   testa = coda = NULL;
   cap = c < 1 ? 1 : c;
   num = 0;
}

void Buffer::copia(const Buffer& s)
{
   num = s.num;
   cap = s.cap:
   testa = coda = NULL;
   if (s.testa != NULL) {
      testa = new elem;
      testa->val = s.testa->val;
      elem *tmp = testa, *aux = s.testa->next;
      while (aux != NULL) {
         tmp->next = new elem;
         tmp = tmp->next;
         tmp->val = aux->val;
         aux = aux->next;
      }
      tmp->next = NULL;
      coda = tmp;
   }
}

void Buffer::elimina()
{
   elem *tmp = testa;
   while (testa != NULL) {
      testa = testa->next;
      delete tmp;
      tmp = testa;
   }
   num = 0;
   coda = NULL;
}

Buffer& Buffer::operator=(const Buffer& s)
{
   if (this != &s) {
```

```
      elimina();
      copia(s);
   }
   return *this;
}

int Buffer::inserisci(int v)
{
   if (num == cap)
      return 0;
   elem *tmp = new elem;
   tmp->val = v;
   tmp->next = NULL;
   if (testa == NULL)
      testa = coda = tmp;
   else {
      coda->next = tmp;
      coda = tmp;
   }
   num++;
   return 1;
}

int Buffer::estrai(int& v)
{
   if (num == 0)
      return 0;
   elem *tmp = testa;
   testa = testa->next;
   v = tmp->val;
   delete tmp;
   if (num == 1)
      coda = NULL;
   num--;
   return 1;
}

int Buffer::capacita(int c)
{
   if (c < num)
      return 0;
   cap = c;
   return 1;
}

int Buffer::operator%(int v) const
{
   int conta = 0;
   for (elem* tmp = testa; tmp != NULL; tmp = tmp->next)
      if (tmp->val == v)
         conta++;
   return conta;
}

ostream& operator<<(ostream& os, const Buffer& s)
{
   os << '[' << s.cap << "] <";
   for (Buffer::elem* aux = s.testa; aux != NULL; aux = aux->next) {
      os << aux->val;
      if (aux != s.coda)
         os << ", ";
   }
```

```
        os << '>';
        return os;
    }
```

Esercizio n. 24 – Deposito

Un *deposito* è in grado di contenere un numero illimitato di elementi aventi valore intero. Le operazioni che possono essere effettuate su un deposito sono le seguenti:

- `Deposito t;`
  Costruttore di default, che inizializza un deposito *t*. Inizialmente, il deposito è vuoto.
- `Deposito t1(t);`
  Costruttore di copia, che inizializza il deposito *t1* col valore del deposito *t*.
- `t1 = t`
  Operatore di assegnamento, che sostituisce il valore del deposito risultato *t1* con quello del deposito *t*.
- `t.inserisci(v)`
  Operazione che inserisce un elemento avente valore *v* nel deposito *t*.
- `t.estrai(sel, v)`
  Operazione che estrae un elemento dal deposito *t*, e ritorna il valore *v* di tale elemento. Se *sel* vale 0, l'elemento estratto è quello inserito nel deposito da più tempo; se *sel* vale 1, l'elemento estratto è quello inserito nel deposito più di recente. L'operazione fallisce se il deposito è vuoto. L'operazione ritorna un'indicazione di successo (1) o fallimento (0).
- `t.valore(sel, v)`
  Operazione che ritorna il valore *v* di un elemento contenuto nel deposito *t*. Se *sel* vale 0, l'elemento è quello avente valore minimo; se *sel* vale 1, l'elemento è quello avente valore massimo. Il deposito resta inalterato. L'operazione fallisce se il deposito è vuoto. L'operazione ritorna un'indicazione di successo (1) o fallimento (0).
- `t % v`
  Operatore di modulo, che ritorna il numero di elementi aventi valore *v* contenuti nel deposito *t*.
- `!t`
  Operatore di negazione logica, che ritorna il numero totale di elementi contenuti nel deposito *t*.
- `~Deposito()`
  Distruttore.

Mediante 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.

Soluzione

```
#include <iostream>
using namespace std;

class Deposito {
   struct elem {
      int val;
      elem *prec, *succ;
   };
   elem *primo, *ultimo;
   int num;
   void copia(const Deposito&);
   void elimina();
public:
   Deposito() { primo = ultimo = NULL; num = 0; }
   Deposito(const Deposito& s) { copia(s); }
   Deposito& operator=(const Deposito&);
```

```
   void inserisci(int);
   int estrai(int, int&);
   int valore(int, int&) const;
   int operator!() const { return num; }
   int operator%(int) const;
   ~Deposito() { elimina(); }
};

void Deposito::copia(const Deposito& s)
{
   primo = ultimo = NULL;
   num = s.num;
   if (s.primo != NULL) {
      primo = new elem;
      primo->val = s.primo->val;
      primo->prec = NULL;
      elem *aux1 = s.primo->succ, *aux2 = primo;
      while (aux1 != NULL) {
         aux2->succ = new elem;
         aux2->succ->prec = aux2;
         aux2 = aux2->succ;
         aux2->val = aux1->val;
         aux1 = aux1->succ;
      }
      ultimo = aux2;
      ultimo->succ = NULL;
   }
}

void Deposito::elimina()
{
   ultimo = primo;
   while (primo != NULL) {
      primo = primo->succ;
      delete ultimo;
      ultimo = primo;
   }
}

Deposito& Deposito::operator=(const Deposito& s)
{
   if (this != &s) {
      elimina();
      copia(s);
   }
   return *this;
}

void Deposito::inserisci(int v)
{
   elem *tmp = new elem;
   tmp->val = v;
   tmp->succ = NULL;
   if (primo == NULL) {
      primo = tmp;
      primo->prec = NULL;
      ultimo = primo;
   }
   else {
      ultimo->succ = tmp;
      tmp->prec = ultimo;
      ultimo = ultimo->succ;
```

```
   }
   num++;
}

int Deposito::estrai(int sel, int& v)
{
   if (num == 0)
      return 0;
   elem *tmp;
   if (sel == 0) { // estrazione in coda
      tmp = ultimo;
      ultimo = ultimo->prec;
      if (ultimo == NULL)
         primo = NULL;
      else
         ultimo->succ = NULL;
   }
   else if (sel == 1) { // estrazione in testa
      tmp = primo;
      primo = primo->succ;
      if (primo == NULL)
         ultimo = NULL;
      else
         primo->prec = NULL;
   }
   else
      return 0;
   num--;
   v = tmp->val;
   delete tmp;
   return 1;
}

int Deposito::valore(int sel, int& v) const
{
   if (num == 0)
      return 0;
   v = primo->val;
   elem* aux = primo->succ;
   if (sel == 0)
      while (aux != NULL) {
         if (aux->val < v)
            v = aux->val;
         aux = aux->succ;
      }
   else if (sel == 1)
      while (aux != NULL) {
         if (aux->val > v)
            v = aux->val;
         aux = aux->succ;
      }
   else
      return 0;
   return 1;
}

int Deposito::operator%(int v) const
{
   int conta = 0;
   elem* aux = primo;
   while (aux != NULL) {
      if (aux->val == v)
```

```
            conta++;
         aux = aux->succ;
      }
      return conta;
   }
```

Esercizio n. 25 – Tavola

Una *tavola* è divisa in caselle organizzate in un ugual numero di righe e di colonne. Il numero delle righe e delle colonne è minore o uguale al numero di bit che formano un *unsigned int*. Ciascuna casella può essere trasparente od opaca. Le operazioni che possono essere effettuate su una tavola sono specificate nel seguito. In alcune operazioni, un *selettore* è usato per selezionare le righe o le colonne della tavola. Il selettore è un *unsigned int* che viene interpretato come vettore di bit, e i bit sono numerati a partire dal meno significativo (il bit 0); il bit *i*-esimo corrisponde alla *i*-esima riga o alla *i*-esima colonna e, se ad 1, specifica che quella riga o colonna è selezionata.

- `Tavola t;`
  Costruttore di default, che inizializza una tavola *t* formata da una sola casella. Inizialmente tale casella è opaca.
- `Tavola t(n);`
  Costruttore che inizializza una tavola *t* formata da *n* righe ed *n* colonne di caselle. Inizialmente tutte le caselle sono opache.
- `Tavola t1(t);`
  Costruttore di copia, che inizializza la tavola *t1* col valore della tavola *t*.
- `t1 = t`
  Operatore di assegnamento, che sostituisce il valore della tavola *t1* con quello della tavola *t*.
- `t.opacheRighe(sel)`
  Operazione che rende opache le caselle delle righe della tavola *t* selezionate dal selettore *sel*.
- `t.opacheColonne(sel)`
  Operazione che rende opache le caselle delle colonne della tavola *t* selezionate dal selettore *sel*.
- `t.trasparentiRighe(sel)`
  Operazione che rende trasparenti le caselle delle righe della tavola *t* selezionate dal selettore *sel*.
- `t.trasparentiColonne(sel)`
  Operazione che rende trasparenti le caselle delle colonne della tavola *t* selezionate dal selettore *sel*.
- `t.quanteOpache()`
  Operazione che ritorna il numero totale di caselle opache presenti nella tavola *t*.
- `t.quanteTrasparenti()`
  Operazione che ritorna il numero totale di caselle trasparenti presenti nella tavola *t*.
- `cout << t`
  Operatore di uscita per il tipo *Tavola*. L'uscita ha la forma seguente:

  ```
  T O T
  T T O
  T O T
  ```

  Il carattere 'T' indica una casella trasparente, il carattere 'O' indica una casella opaca.
- `~Tavola()`
  Distruttore.

Mediante il linguaggio C++, realizzare il tipo di dati astratti *Tavola* definito dalle precedenti specifiche. Utilizzare una rappresentazione interna della tavola che renda minima l'occupazione di memoria. Individuare le eventuali situazioni di errore, e metterne in opera un corretto trattamento.

Soluzione

```
#include <iostream>
using namespace std;

const int SIZE = sizeof(unsigned int) * 8;

class Tavola {
   friend ostream& operator<<(ostream&, const Tavola&);
   unsigned int* tavola;
   int dim;  // opaco: 0, trasparente: 1
public:
   Tavola(int = 1);
   Tavola(const Tavola&);
   Tavola& operator=(const Tavola&);
   Tavola& opacheRighe(unsigned int);
   Tavola& opacheColonne(unsigned int);
   Tavola& trasparentiRighe(unsigned int);
   Tavola& trasparentiColonne(unsigned int);
   int quanteOpache() const { return dim * dim - quanteTrasparenti(); }
   int quanteTrasparenti() const;
   ~Tavola() { delete[] tavola; }
};

Tavola::Tavola(int n)
{
   dim = (n <= 0) ? 1 : ( (n <= SIZE) ? n : SIZE );
   tavola = new unsigned int[dim];
   for (int i = 0; i < dim; i++)
      tavola[i] = 0;
}

Tavola::Tavola(const Tavola& s)
{
   dim = s.dim;
   tavola = new unsigned int[dim];
   for (int i = 0; i < dim; i++)
      tavola[i] = s.tavola[i];
}

Tavola& Tavola::operator=(const Tavola& s)
{
   if (&s != this) {
      if (dim != s.dim) {
         delete[] tavola;
         dim = s.dim;
         tavola = new unsigned int[dim];
      }
      for (int i = 0; i < dim; i++)
         tavola[i] = s.tavola[i];
   }
   return *this;
}

Tavola& Tavola::opacheRighe(unsigned int sel)
{
   for (int i = 0; i < dim; i++)
      if (sel & (1 << i))
         tavola[i] = 0;
}

Tavola& Tavola::opacheColonne(unsigned int sel)
```

```
{
   for (int i = 0; i < dim; i++)
      tavola[i] &= ~sel;
}

Tavola& Tavola::trasparentiRighe(unsigned int sel)
{
   for (int i = 0; i < dim; i++)
      if (sel & (1 << i))
         tavola[i] = ~0;
}

Tavola& Tavola::trasparentiColonne(unsigned int sel)
{
   for (int i = 0; i < dim; i++)
      tavola[i] |= sel;
}

int Tavola::quanteTrasparenti()
{
   int conta = 0;
   for (int i = 0; i < dim; i++)  // riga
      for (int j = 0; j < dim; j++)  // colonna nella riga
         if ((tavola[i] & (1 << j)))
            conta++;
   return conta;
}

ostream& operator<<(ostream& os, const Tavola& s)
{
   for (int i = 0; i < s.dim; i++) {  // riga
      for (int j = s.dim - 1; j >= 0; j--)  // colonna nella riga
         if (s.tavola[i] & (1 << j))
            os << "\tT";
         else
            os << "\tO";
      os << "\n";
   }
   return os;
}
```

Esercizio n. 26 – Recipiente

Un *recipiente* è in grado di contenere elementi aventi valore intero positivo o nullo. Le operazioni che possono essere effettuate su un recipiente sono le seguenti:

- `Recipiente t;`
  Costruttore di default, che inizializza un recipiente *t* in grado di contenere elementi aventi valore 0. Inizialmente, il recipiente è vuoto.
- `Recipiente t(n);`
  Costruttore che inizializza un recipiente *t* in grado di contenere elementi aventi valore intero compreso tra 0 ed *n*. Inizialmente, il recipiente è vuoto.
- `Recipiente t1(t);`
  Costruttore di copia, che inizializza il recipiente *t1* col valore del recipiente *t*.
- `t1 = t`
  Operatore di assegnamento, che sostituisce il valore del recipiente *t1* con quello del recipiente *t*.
- `t.inserisci(v)`
  Operazione che inserisce un elemento avente valore *v* nel recipiente *t*.
- `t.estrai(v, m)`
  Operazione che estrae *m* elementi aventi valore *v* dal recipiente *t*. Se il recipiente *t* contiene meno di *m* elementi aventi valore *v*, l'operazione estrae tutti questi elementi. L'operazione ritorna il numero di elementi effettivamente estratti dal recipiente.
- `t.estraiRecente(m)`
  Operazione che estrae *m* elementi dal recipiente *t*. Tali elementi hanno il valore dell'elemento inserito più di recente nel recipiente. Se il recipiente *t* contiene meno di *m* elementi aventi tale valore, l'operazione estrae tutti questi elementi. L'operazione ritorna il numero di elementi effettivamente estratti dal recipiente.
- `t[v]`
  Operatore di indicizzazione, che ritorna il numero di elementi aventi valore *v* contenuti nel recipiente *t*.
- `!t`
  Operatore di negazione logica, che ritorna il numero totale di elementi contenuti nel recipiente *t*.
- `cout << t`
  Operatore di uscita per il tipo *Recipiente*. L'uscita ha la forma seguente:

  ```
  <3, 0, 4>
  ```

  In questo esempio, il recipiente è in grado di contenere elementi aventi valore compreso tra 0 ed 2, ed effettivamente contiene tre elementi aventi valore 0, nessun elemento avente valore 1, e quattro elementi aventi valore 2.
- `~Recipiente()`
  Distruttore.

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

Soluzione

```
#include <iostream>
using namespace std;

class Recipiente {
```

```
   friend ostream& operator<<(ostream&, const Recipiente&);
   int *rec;
   int dim;
   int recente;
public:
   Recipiente(int = 0);
   Recipiente(const Recipiente&);
   Recipiente& operator=(const Recipiente&);
   void inserisci(int);
   int estrai(int, int);
   int estraiRecente(int m) { return estrai(recente, m); }
   int operator[](int) const;
   int operator!() const;
   ~Recipiente() { delete[] rec; }
};

Recipiente::Recipiente(int n)
{
   recente = -1;
   dim = (n >= 0) ? n + 1 : 1;
   rec = new int[dim];
   for (int i = 0; i < dim; i++)
      rec[i] = 0;
}

Recipiente::Recipiente(const Recipiente& s)
{
   dim = s.dim;
   recente = s.recente;
   rec = new int[dim];
   for (int i = 0; i < dim; i++)
      rec[i] = s.rec[i];
}

Recipiente& Recipiente::operator=(const Recipiente& s)
{
   if (this != &s) {
      if (dim != s.dim) {
         delete[] rec;
         dim = s.dim;
         rec = new int[dim];
      }
      recente = s.recente;
      for (int i = 0; i < dim; i++)
         rec[i] = s.rec[i];
   }
   return *this;
}

void Recipiente::inserisci(int v)
{
   if (v >= 0 && v < dim)
      rec[recente = v]++;
}

int Recipiente::estrai(int v, int m)
{
   int estratti = 0;
   if (v >= 0 && v < dim && m > 0)
      if (rec[v] >= m) {
         estratti = m;
         rec[v] -= m;
```

```
      }
      else {
         estratti = rec[v];
         rec[v] = 0;
      }
   return estratti;
}

int Recipiente::operator[](int v) const
{
   if (v >= 0 && v < dim)
      return rec[v];
   return 0;
}

int Recipiente::operator!() const
{
   int somma = 0;
   for (int i = 0; i < dim; i++)
      somma += rec[i];
   return somma;
}

ostream& operator<<(ostream& os, const Recipiente& s)
{
   os << '<';
   for (int i = 0; i < s.dim; i++) {
      os << s.rec[i];
      if (i < (s.dim - 1))
         os << ', ';
   }
   os << '>';
   return os;
}
```

Esercizio n. 27 – Piatto

Lungo il perimetro di un *piatto* sono disposte cavità contrassegnate dalle lettere maiuscole dell'alfabeto, a partire dalla lettera 'A'. Ciascuna cavità può essere libera oppure occupata da una sfera. Le sfere hanno un valore intero maggiore di 0. Le operazioni che possono essere effettuate su un piatto sono le seguenti:

- `Piatto t;`
  Costruttore di default, che inizializza un piatto *t* avente una sola cavità. Inizialmente, tale cavità è libera.
- `Piatto t(n);`
  Costruttore che inizializza un piatto *t* avente *n* cavità, con $n \leq 26$. Inizialmente, tutte le cavità sono libere.
- `Piatto t1(t);`
  Costruttore di copia, che inizializza il piatto *t1* col valore del piatto *t*.
- `t1 = t`
  Operatore di assegnamento, che sostituisce il valore del piatto *t1* con quello del piatto *t*.
- `t.inserisci(c, v)`
  Operazione che inserisce una sfera avente valore *v* nella cavità contrassegnata dalla lettera *c* del piatto *t*. L'operazione fallisce se la cavità *c* è già occupata. L'operazione ritorna un'indicazione di successo (1) o fallimento (0).
- `t.estrai(c)`
  Operazione che estrae la sfera che occupa la cavità *c* del piatto *t*. L'operazione ritorna il valore della sfera estratta. Se la cavità *c* è libera, l'operazione ritorna 0.
- `t.estraiMultiplo(c)`
  Operazione che estrae dal piatto *t* le sfere contenute nella cavità *c* ed in tutte le cavità occupate ad essa adiacenti, fino alla prima cavità libera in ambedue le direzioni (cioè contrassegnate sia da lettere crescenti sia da lettere decrescenti). L'adiacenza è circolare, cioè l'ultima cavità del piatto è considerata adiacente alla prima. L'operazione ritorna la somma dei valori di tali sfere. Se la cavità *c* è libera, l'operazione ritorna 0.
- `t[c]`
  Operatore di indicizzazione, che ritorna il valore della sfera che occupa la cavità *c* del piatto *t*. Se tale cavità è libera, l'operazione ritorna 0.
- `!t`
  Operatore di negazione logica, che ritorna la somma dei valori di tutte le sfere che occupano le cavità del piatto *t*. Se tutte le cavità sono libere, l'operazione ritorna 0.
- `cout << t`
  Operatore di uscita per il tipo *Piatto*. L'uscita ha la forma seguente:

  ```
  <-, 3, -, 1, 1, 10>
  ```

  In questo esempio, il piatto *t* ha 6 cavità. La prima cavità, contrassegnata dalla lettera 'A', è libera. L'ultima cavità, contrassegnata dalla lettera 'F', è occupata da una sfera avente valore 10.
- `~Piatto()`
  Distruttore.

Mediante il linguaggio C++, realizzare il tipo di dati astratti *Piatto* definito dalle precedenti specifiche. Fare ricorso ad una rappresentazione tesa a ridurre l'occupazione di memoria per piatti con poche cavità. Individuare le eventuali situazioni di errore, e metterne in opera un corretto trattamento.

Soluzione

```
#include <iostream>
using namespace std;

class Piatto {
   friend ostream& operator<<(ostream&, const Piatto&);
   int dim;
   int *ptt;
public:
   Piatto(int n = 1);
   Piatto(const Piatto&);
   Piatto& operator=(const Piatto&);
   int inserisci(char, int);
   int estrai(char);
   int estraiMultiplo(char);
   int operator[](char c) const {
      return ( (c - 'A' < 0) || (c - 'A' >= dim) ) ? 0 : ptt[c - 'A']; }
   int operator!() const;
   ~Piatto() { delete[] ptt; }
};

Piatto::Piatto(int n)
{
   dim = (n < 1) ? 1 : (n > 26 ? 26 : n);
   ptt = new int[dim];
   for (int i = 0; i < dim; i++)
      ptt[i] = 0;
}

Piatto::Piatto(const Piatto& s)
{
   dim = s.dim;
   ptt = new int[dim];
   for (int i = 0; i < dim; i++)
      ptt[i] = s.ptt[i];
}

Piatto& Piatto::operator=(const Piatto& s)
{
   if (this != &s) {
      if (dim != s.dim) {
         dim = s.dim;
         delete[] ptt;
         ptt = new int[dim];
      }
      for (int i = 0; i < dim; i++)
         ptt[i] = s.ptt[i];
   }
   return *this;
}

int Piatto::inserisci(char c, int v)
{
   int indice = c - 'A';
   if (indice < 0 || indice >= dim || ptt[indice] != 0)
      return 0;
   ptt[indice] = v;
   return 1;
}

int Piatto::estrai(char c)
```

```
{
   int indice = c - 'A';
   if (indice < 0 || indice >= dim || ptt[indice] == 0)
      return 0;
   int v = ptt[indice];
   ptt[indice] = 0;
   return v;
}

int Piatto::estraiMultiplo(char c)
{
   int indice = c - 'A';
   if (indice < 0 || indice >= dim || ptt[indice] == 0)
      return 0;
   int v = 0;
   int tmp = (indice + 1) < dim ? (indice + 1) : 0;
   while (ptt[tmp] != 0) {
      v += ptt[tmp];
      ptt[tmp] = 0;
      tmp = (tmp + 1) < dim ? (tmp + 1) : 0;
   }
   tmp = indice;
   while (ptt[tmp] != 0) {
      v += ptt[tmp];
      ptt[tmp] = 0;
      tmp = (tmp - 1) < 0 ? (dim - 1) : (tmp - 1);
   }
   return v;
}

int Piatto::operator!() const
{
   int conta = 0;
   for (int i = 0; i < dim; i++)
      conta += ptt[i];
   return conta;
}

ostream& operator<<(ostream& os, const Piatto& s)
{
   os << '<';
   if (s.ptt[0] == 0)
      os << '-';
   else
      os << s.ptt[0];
   for (int i = 1; i < s.dim; i++)
      if (s.ptt[i] == 0)
         os << ", -";
      else
         os << ", " << s.ptt[i];
   os << '>';
   return os;
}
```

Esercizio n. 28 – Barra

Una *barra* è formata da segmenti di ferro e di rame. Le posizioni dei segmenti nella barra sono numerate a partire da 1. Le operazioni che possono essere effettuate su una barra sono le seguenti:

- `Barra t;`
  Costruttore di default, che inizializza una barra *t*. Tale barra può essere formata da un numero illimitato di segmenti di ferro, e da al più un segmento di rame. Inizialmente la barra è formata da un solo segmento, e tale segmento è di rame.
- `Barra t(n);`
  Costruttore che inizializza una barra *t*. Tale barra può essere formata da un numero illimitato di segmenti di ferro, e da al più *n* segmenti di rame. Inizialmente la barra è formata da un solo segmento, e tale segmento è di rame.
- `Barra t1(t);`
  Costruttore di copia, che inizializza la barra *t1* col valore della barra *t*.
- `t1 = t`
  Operatore di assegnamento, che sostituisce il valore della barra *t1* con quello della barra *t*.
- `t.aggiungi(m)`
  Operazione che aggiunge un segmento di metallo *m* (ferro o rame) in fondo alla barra *t*. L'operazione fallisce se si vuole aggiungere un segmento di rame e la barra già contiene il numero massimo di segmenti di rame. L'operazione ritorna un'indicazione di successo (1) o fallimento (0).
- `t.elimina(pos, m)`
  Operazione che elimina un segmento dalla barra *t*. L'argomento *pos* rappresenta la posizione di tale segmento nella barra. L'operazione ritorna il metallo *m* del segmento eliminato (ferro o rame). L'operazione fallisce se la barra è formata da un solo segmento, oppure *pos* è maggiore del numero totale dei segmenti nella barra. L'operazione ritorna un'indicazione di successo (1) o fallimento (0).
- `t[pos]`
  Operatore di indicizzazione, che ritorna il metallo del segmento in posizione *pos* della barra *t*. Se *pos* è maggiore del numero totale di segmenti della barra, l'operazione ritorna il metallo del segmento in fondo alla barra.
- `t % m`
  Operatore di modulo, che ritorna il numero dei segmenti di metallo *m* che formano la barra *t*.
- `!t`
  Operatore di negazione logica, che ritorna il numero totale dei segmenti che formano la barra *t*.
- `cout << t`
  Operatore di uscita per il tipo *Barra*. L'uscita ha la forma seguente:

  ```
  FFRFFR
  ```

  In questo esempio, la barra *t* è formata da sei segmenti. Il segmento in posizione 1, in cima alla barra, è di ferro. Il segmento in posizione 6, in fondo alla barra, è di rame.
- `~Barra()`
  Distruttore.

Mediante il linguaggio C++, realizzare il tipo di dati astratti *Barra* definito dalle precedenti specifiche. *Fare ricorso ad una rappresentazione a vettore.* Individuare le eventuali situazioni di errore, e metterne in opera un corretto trattamento

Soluzione

```
#include <iostream>
using namespace std;

enum metallo { FERRO, RAME };

class Barra {
   friend ostream& operator<<(ostream&, const Barra&);
   int max_rame, num_rame, num_ferro;
   int *br;
public:
   Barra(int n = 1);
   Barra(const Barra&);
   Barra &operator=(const Barra&);
   int aggiungi(metallo);
   int elimina(int, metallo&);
   metallo operator[](int) const;
   int operator%(metallo) const;
   int operator!() const { return num_rame + num_ferro; }
   ~Barra() { delete[] br; }
};

Barra::Barra(int n)
{
   max_rame = (n < 1) ? 1 : n;
   num_ferro = 0;
   num_rame = 1;
   br = new int[max_rame];
   br[0] = 1;
   for (int i = 1; i < max_rame; i++)
      br[i] = 0;
}

Barra::Barra(const Barra& s)
{
   max_rame = s.max_rame;
   num_ferro = s.num_ferro;
   num_rame = s.num_rame;
   br = new int[max_rame];
   for (int i = 0; i < max_rame; i++)
      br[i] = s.br[i];
}

Barra& Barra::operator=(const Barra& s)
{
   if (this != &s) {
      if (max_rame != s.max_rame) {
         max_rame = s.max_rame;
         delete[] br;
         br = new int[max_rame];
      }
      num_ferro = s.num_ferro;
      num_rame = s.num_rame;
      for (int i = 0; i < max_rame; i++)
         br[i] = s.br[i];
   }
   return *this;
}

int Barra::aggiungi(metallo m)
{
```

```
   int posizione_corrente = num_rame + num_ferro + 1;
   if (m == RAME) {
      if (num_rame == max_rame)
         return 0;
      br[num_rame++] = posizione_corrente;
   }
   else
      num_ferro++;
   return 1;
}

int Barra::elimina(int pos, metallo& m)
{
   if ( num_rame + num_ferro <= 1 || pos > num_rame + num_ferro )
      return 0;
   int j;
   for (j = 0; j < num_rame; j++)
      if (pos == br[j])
         break;
   if (j < num_rame) {
      m = RAME;
      for (; j < num_rame - 1; j++)
         br[j] = br[j + 1];
      num_rame--;
   }
   else {
      m = FERRO;
      num_ferro--;
   }
   for (int i = 0; i < num_rame; i++)
      if (br[i] > pos)
         br[i]--;
   return 1;
}

metallo Barra::operator[](int pos) const
{
   int totale = num_rame + num_ferro;
   pos = (pos > totale) ? totale : pos;
   for (int i = 0; i < num_rame; i++)
      if (pos == br[i])
         return RAME;
   return FERRO;
}

int Barra::operator%(metallo m) const
{
   if (m == RAME)
      return num_rame;
   return num_ferro;
}

ostream& operator<<(ostream& os, const Barra& s)
{
   int totale = s.num_rame + s.num_ferro;
   int contatore_rame = 0;
   for (int i = 1; i <= totale; i++)
      if ( (contatore_rame < s.num_rame)
           && (i == s.br[contatore_rame]) ) {
         cout << 'R';
         contatore_rame++;
      }
```

```
        else
            cout << 'F';
        os << '\n';
        return os;
    }
```

Esercizio n. 29 – Sistema di code

Un *sistema di code* è formato da code di elementi aventi valore intero strettamente positivo. Le code hanno capacità illimitata e sono numerate a partire da 1. Le operazioni che possono essere effettuate su un sistema di code sono le seguenti:

- `SistemaDiCode t;`
  Costruttore di default, che inizializza un sistema di code *t* formato da una sola coda. Inizialmente, la coda è vuota.
- `SistemaDiCode t(n);`
  Costruttore che inizializza un sistema di code *t* formato da *n* code. Inizialmente, le code sono vuote.
- `SistemaDiCode t1(t);`
  Costruttore di copia, che inizializza il sistema di code *t1* col valore del sistema di code *t*.
- `t1 = t`
  Operatore di assegnamento, che sostituisce il valore del sistema di code risultato *t1* con quello del sistema di code *t*.
- `t.inserisci(v)`
  Operazione che inserisce un elemento avente valore *v* nel sistema di code *t*. L'elemento viene aggiunto in fondo alla coda contenente il minor numero di elementi.
- `t.inserisci(c, v)`
  Operazione che inserisce un elemento avente valore *v* in fondo alla coda *c* del sistema di code *t*.
- `t.estrai()`
  Operazione che estrae un elemento dal sistema di code *t*, e ritorna il valore di tale elemento. L'elemento viene estratto dalla cima della coda contenente il maggior numero di elementi. L'operazione fallisce se tutte le code del sistema di code *t* sono vuote. In caso di fallimento l'operazione ritorna 0.
- `t.estrai(c)`
  Operazione che estrae un elemento dalla cima della coda *c* del sistema di code *t*, e ritorna il valore di tale elemento. L'operazione fallisce se la coda *c* è vuota. In caso di fallimento l'operazione ritorna 0.
- `~SistemaDiCode()`
  Distruttore.

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

Soluzione

```
#include <iostream>
using namespace std;

class SistemaDiCode {
   class Coda {
      struct elem {
         int val;
         elem *succ;
      };
      int elementi;
      elem *primo, *ultimo;
      void copia(const Coda&);
      void elimina();
   public:
```

```
      Coda() { primo = ultimo = NULL; elementi = 0; }
      Coda(const Coda& s) { copia(s); }
      Coda& operator=(const Coda&);
      void inserisci(int);
      int estrai(int&);
      int operator!() const { return elementi; }
      ~Coda() { elimina(); }
   };
   Coda *code;
   int dim;
public:
   SistemaDiCode(int = 1);
   SistemaDiCode(SistemaDiCode& s);
   SistemaDiCode& operator=(const SistemaDiCode&);
   void inserisci(int);
   void inserisci(int, int);
   int estrai();
   int estrai(int);
   ~SistemaDiCode() { delete[] code; }
};

// CODA

void SistemaDiCode::Coda::copia(const Coda& s)
{
   primo = ultimo = NULL;
   elementi = s.elementi;
   if (s.primo != NULL) {
      primo = new elem;
      primo->val = s.primo->val;
      ultimo = primo;
      elem *aux = s.primo->succ;
      while (aux != NULL) {
         ultimo->succ = new elem;
         ultimo = ultimo->succ;
         ultimo->val = aux->val;
         aux = aux->succ;
      }
      ultimo->succ = NULL;
   }
}

void SistemaDiCode::Coda::elimina()
{
   ultimo = primo;
   while (primo != NULL) {
      primo = primo->succ;
      delete ultimo;
      ultimo = primo;
   }
}

SistemaDiCode::Coda& SistemaDiCode::Coda::operator=(const Coda& s)
{
   if (this != &s) {
      elimina();
      copia(s);
   }
   return *this;
}

void SistemaDiCode::Coda::inserisci(int d)
```

```
{
   if (primo == NULL) {
      primo = new elem;
      ultimo = primo;
      primo->val = d;
      primo->succ = NULL;
   }
   else {
      ultimo->succ = new elem;
      ultimo = ultimo->succ;
      ultimo->val = d;
      ultimo->succ = NULL;
   }
   elementi += 1;
}

int SistemaDiCode::Coda::estrai(int &v)
{
   if (primo != NULL) {
      elem *tmp = primo;
      primo = primo->succ;
      v = tmp->val;
      delete tmp;
      elementi -= 1;
      return 1;
   }
   return 0;
}

// SISTEMA DI CODE

SistemaDiCode::SistemaDiCode(int n)
{
   dim = (n > 0) ? n : 1;
   code = new Coda[dim];
}

SistemaDiCode::SistemaDiCode(SistemaDiCode& s)
{
   dim = s.dim;
   code = new Coda[dim];
   for (int i = 0; i < dim; i++)
      code[i] = s.code[i];
}

SistemaDiCode& SistemaDiCode::operator=(const SistemaDiCode& s)
{
   if (this != &s) {
      if (dim != s.dim) {
         dim = s.dim;
         delete [] code;
         code = new Coda[dim];
      }
      for (int i = 0; i < dim; i++)
         code[i] = s.code[i];
   }
   return *this;
}

void SistemaDiCode::inserisci(int v)
{
   int minimo = !code[0];
```

```
   int minimo_indice = 0;
   for (int i = 1; i < dim; i++)
      if (!code[i] < minimo) {
         minimo = !code[i];
         minimo_indice = i;
      }
   code[minimo_indice].inserisci(v);
}

void SistemaDiCode::inserisci(int v, int c)
{
   int indice = c < 1 ? 1 : (c > dim ? dim : c);
   code[indice - 1].inserisci(v);
}

int SistemaDiCode::estrai()
{
   int massimo = !code[0];
   int massimo_indice = 0;
   for (int i = 1; i < dim; i++)
      if (!code[i] > massimo) {
         massimo = !code[i];
         massimo_indice = i;
      }
   if (massimo == 0)
      return 0;
   int valore;
   code[massimo_indice].estrai(valore);
   return valore;
}

int SistemaDiCode::estrai(int c)
{
   int indice = c < 1 ? 1 : (c > dim ? dim : c);
   if (!code[indice - 1] == 0)
      return 0;
   int valore;
   code[indice - 1].estrai(valore);
   return valore;
}
```

Esercizio n. 30 – Catena

Una *catena* è formata da un numero illimitato di anelli di metalli diversi. I possibili metalli sono il rame, l'argento e l'oro. Le operazioni che possono essere effettuate su una catena sono le seguenti:

- `Catena t;`
  Costruttore di default, che inizializza una catena *t*. Inizialmente la catena è vuota.
- `Catena t1(t);`
  Costruttore di copia, che inizializza la catena *t1* col valore della catena *t*.
- `t1 = t`
  Operatore di assegnamento, che sostituisce il valore della catena *t1* con quello della catena *t*.
- `t.inserisci(v)`
  Operazione che inserisce un anello di metallo *v* in fondo alla catena *t*.
- `t.estrai()`
  Operazione che estrae il primo anello della catena *t*. L'operazione fallisce se la catena è vuota. L'operazione ritorna un'indicazione di successo (1) o fallimento (0).
- `t.estrai(v)`
  Operazione che estrae dalla catena *t* il primo anello di metallo *v*. L'operazione fallisce se la catena non contiene anelli di metallo *v*. L'operazione ritorna un'indicazione di successo (1) o fallimento (0).
- `t % v`
  Operatore di modulo, che ritorna il numero di anelli di metallo *v* contenuti nella catena *t*.
- `!t`
  Operatore di negazione logica, che ritorna il numero totale di anelli che formano la catena *t*.
- `cout << t`
  Operatore di uscita per il tipo *Catena*. L'uscita ha la forma seguente:

  ```
  RAAROO
  ```

  La lettera R indica un anello di rame, la lettera A indica un anello d'argento e la lettera O indica un anello d'oro. In questo esempio, la catena *t* è formata da sei anelli, il primo anello è di rame e l'ultimo d'oro.
- `~Catena()`
  Distruttore.

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

Soluzione

```
#include <iostream>
using namespace std;

enum metallo { RAME, ARGENTO, ORO };

class Catena {
   friend ostream &operator<<(ostream &os, const Catena&);
   struct elem {
      metallo val;
      elem *next;
   };
   elem *testa;
```

```
   elem *coda;
   int num;
   void copia(const Catena&);
   void elimina();
public:
   Catena() { testa = coda = NULL; num = 0; }
   Catena(const Catena& s) { copia(s); }
   Catena& operator=(const Catena&);
   void inserisci(metallo);
   int estrai();
   int estrai(metallo);
   int operator%(metallo) const;
   int operator!() const { return num; }
   ~Catena() { elimina(); }
};

void Catena::copia(const Catena& s)
{
   num = 0;
   testa = coda = NULL;
   if (s.testa != NULL) {
      testa = new elem;
      testa->val = s.testa->val;
      num++;
      elem *tmp = testa, *aux = s.testa->next;
      while (aux != NULL) {
         tmp->next = new elem;
         tmp = tmp->next;
         tmp->val = aux->val;
         aux = aux->next;
         num++;
      }
      coda = tmp;
      tmp->next = NULL;
   }
}

void Catena::elimina()
{
   elem *tmp = testa;
   while (testa != NULL) {
      testa = testa->next;
      delete tmp;
      tmp = testa;
   }
}

Catena& Catena::operator=(const Catena& s)
{
   if (this != &s) {
      elimina();
      copia(s);
   }
   return *this;
}

void Catena::inserisci(metallo v)
{
   elem *tmp = new elem;
   tmp->val = v;
   tmp->next = NULL;
   if (num == 0)
```

```
      testa = coda = tmp;
   else {
      coda->next = tmp;
      coda = coda->next;
   }
   num++;
}

int Catena::estrai()
{
   if (num == 0)
      return 0;
   elem *tmp = testa;
   testa = testa->next;
   delete tmp;
   num--;
   return 1;
}

int Catena::estrai(metallo v)
{
   if (testa == NULL)
      return 0;
   elem *tmp, *prec;
   for (tmp = testa; tmp != NULL && tmp->val != v; tmp = tmp->next)
      prec = tmp;
   if (tmp == NULL)
      return 0;
   if (tmp == testa)
      testa = tmp->next;
   else
      prec->next = tmp->next;
   delete tmp;
   num--;
   return 1;
}

int Catena::operator%(metallo v) const
{
   int totale = 0;
   for (elem *aux = testa; aux != NULL; aux = aux->next)
      if (aux->val == v)
         totale++;
   return totale;
}

ostream &operator<<(ostream &os, const Catena& s)
{
   Catena::elem *tmp;
   for (tmp = s.testa; tmp != NULL; tmp = tmp->next) {
      if (tmp->val == RAME)
         os << 'R';
      else if (tmp->val == ARGENTO)
         os << 'A';
      else
         os << 'O';
   }
   return os;
}
```

Esercizio n. 31 – Lista colorata

Una *lista colorata* è formata da elementi colorati aventi valore intero. I possibili colori sono il rosso, il giallo e il verde. Le operazioni che possono essere effettuate su una lista colorata sono le seguenti:

- `ListaColorata t;`
  Costruttore di default, che inizializza una lista colorata *t*. Inizialmente, la lista è vuota.
- `ListaColorata t1(t);`
  Costruttore di copia, che inizializza la lista colorata *t1* col valore della lista colorata *t*.
- `t1 = t`
  Operatore di assegnamento, che sostituisce il valore della lista colorata *t1* con quello della lista colorata *t*.
- `t.inserisci(c, v)`
  Operazione che inserisce un elemento avente colore *c* e valore *v* nella lista colorata *t*.
- `t.estrai(c, v)`
  Operazione che estrae l'elemento di colore *c* inserito da più tempo nella lista colorata *t*, e ritorna il valore *v* di tale elemento. L'operazione fallisce se la lista colorata *t* non contiene elementi di colore *c*. L'operazione ritorna un'indicazione di successo (1) o fallimento (0).
- `t[c]`
  Operatore di indicizzazione, che ritorna la somma dei valori degli elementi di colore *c* contenuti nella lista colorata *t*.
- `t % c`
  Operatore di modulo, che ritorna il numero di elementi di colore *c* contenuti nella lista colorata *t*.
- `!t`
  Operatore di negazione logica, che ritorna il numero totale di elementi contenuti nella lista colorata *t*.
- `cout << t`
  Operatore di uscita per il tipo *ListaColorata*. L'uscita ha la forma seguente:

  ```
  <10, 0, 2>
  ```

  In questo esempio, la lista colorata *t* contiene 10 elementi rossi, nessun elemento giallo, e 2 elementi verdi.
- `~ListaColorata()`
  Distruttore.

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

Soluzione

```
#include <iostream>
using namespace std;

enum colore {ROSSO, GIALLO, VERDE};
const int NUM_COLORI = 3;

class ListaColorata {
   friend ostream& operator<<(ostream& os, const ListaColorata&);
   class Lista {
      struct elem {
         int val;
         elem *next;
```

```
      };
      elem *testa, *coda;
      int num;
      void copia(const Lista&);
      void elimina();
   public:
      Lista() : testa(NULL), coda(NULL), num(0) {}
      Lista(const Lista& s) { copia(s); }
      Lista& operator=(const Lista&);
      void inserisci(int);
      int estrai(int&);
      int somma() const;
      int operator!() const { return num; }
      ~Lista() { elimina(); }
   };
   Lista liste[NUM_COLORI];
public:
   ListaColorata() {}
// ListaColorata(const ListaColorata&);
// ListaColorata operator=(const ListaColorata&);
   void inserisci(colore, int);
   int estrai(colore, int&);
   int operator[](colore) const;
   int operator%(colore) const;
   int operator!() const;
// ~ListaColorata();
};

// ----------- Lista -----------

void ListaColorata::Lista::copia(const ListaColorata::Lista& s)
{
   num = s.num;
   testa = NULL;
   coda = NULL;
   if (s.testa != NULL) {
      testa = new elem;
      testa->val = s.testa->val;
      elem *tmp = testa, *aux = s.testa->next;
      while (aux != NULL) {
         tmp->next = new elem;
         tmp = tmp->next;
         tmp->val = aux->val;
         aux = aux->next;
      }
      coda = tmp;
      tmp->next = NULL;
   }
}

void ListaColorata::Lista::elimina()
{
   elem *tmp = testa;
   while (testa != NULL) {
      testa = testa->next;
      delete tmp;
      tmp = testa;
   }
   coda = NULL;
   num = 0;
}
```

```
ListaColorata::Lista&
ListaColorata::Lista::operator=(const ListaColorata::Lista& s) {
   if (this != &s) {
      elimina();
      copia(s);
   }
   return *this;
}

void ListaColorata::Lista::inserisci(int v)
{ // inserzione in fondo
   elem *tmp = new elem;
   tmp->val = v;
   tmp->next = NULL;
   if (num == 0) {
      testa = tmp;
      coda = tmp;
   }
   else {
      coda->next = tmp;
      coda = coda->next;
   }
   num++;
}

int ListaColorata::Lista::estrai(int &v)
{ // estrazione in testa
   if (num == 0)
      return 0;
   elem *tmp = testa;
   testa = testa->next;
   if (num == 1)
      coda = NULL;
   v = tmp->val;
   delete tmp;
   num--;
   return 1;
}

int ListaColorata::Lista::somma() const
{
   int somma = 0;
   elem *aux = testa;
   while (aux != NULL) {
      somma += aux->val;
      aux = aux->next;
   }
   return somma;
}

// ----------- Lista colorata -----------

inline void ListaColorata::inserisci(colore c, int v)
{
   liste[c].inserisci(v);
}

inline int ListaColorata::estrai(colore c, int &v)
{
   return liste[c].estrai(v);
}
```

```
inline int ListaColorata::operator[](colore c) const
{
   return liste[c].somma();
}

inline int ListaColorata::operator%(colore c) const
{
   return !(liste[c]);
}

int ListaColorata::operator!() const
{
   int somma = 0;
   for (int indice = 0; indice < NUM_COLORI; indice++)
      somma += !liste[indice];
   return somma;
}

ostream& operator<<(ostream& os, const ListaColorata& s)
{
   os << '<' << s % ROSSO << ", " << s % GIALLO << ", "
   os << s % VERDE << '>';
   return os;
}
```

Esercizio n. 32 – Cilindro

Un *cilindro* è in grado di contenere un numero limitato di sfere (la *capacità* del cilindro). Ciascuna sfera ha un valore rappresentato da un numero intero maggiore di 0. Il cilindro è mantenuto in posizione orizzontale. Le sfere possono essere inserite nel cilindro dal lato sinistro o dal lato destro. Quando il cilindro è pieno, l'inserzione di un'ulteriore sfera da un dato lato del cilindro provoca la fuoriuscita di una sfera dal lato opposto. Le operazioni che possono essere effettuate su un cilindro sono le seguenti:

- `Cilindro c;`
  Costruttore di default, che inizializza un cilindro *c* avente capacità unitaria. Inizialmente, il cilindro è vuoto.
- `Cilindro c(n);`
  Costruttore che inizializza un cilindro *c* avente capacità pari a *n* sfere. Inizialmente il cilindro è vuoto.
- `Cilindro c1(c);`
  Costruttore di copia, che inizializza il cilindro *c1* col valore del cilindro *c*.
- `c1 = c`
  Operatore di assegnamento, che sostituisce il valore del cilindro risultato *c1* con quello del cilindro *c*.
- `c.inserisci(t, v)`
  Operazione che inserisce una sfera avente valore *v* dal lato *t* (sinistro o destro) del cilindro *c*. Se il cilindro è pieno, l'inserzione provoca la fuoriuscita dal cilindro di una sfera dal lato opposto. L'operazione ritorna il valore di tale sfera. Se il cilindro non è pieno, l'operazione ritorna 0.
- `c.capacita(p)`
  Operazione che modifica la capacità del cilindro *t*. L'argomento *p* indica la nuova capacità. L'operazione fallisce se la nuova capacità è inferiore al numero di sfere contenute nel cilindro, nel qual caso l'operazione lascia il cilindro inalterato. L'operazione ritorna un'indicazione di successo (1) o fallimento (0).
- `!c`
  Operatore di negazione logica, che ritorna il numero di sfere contenute nel cilindro *c*.
- `cout << c`
  Operatore di uscita per il tipo *Cilindro*. L'uscita ha la forma seguente:

  ```
  (10, 5)
  ```

  In questo esempio, il cilindro *c* ha capacità pari a dieci sfere e attualmente contiene cinque sfere.
- `~Cilindro()`
  Distruttore.

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

Soluzione

```
#include <iostream>
using namespace std;

enum lato {SINISTRO, DESTRO};

class Cilindro {
   friend ostream& operator<<(ostream&, const Cilindro&);
```

```
   int *sfere;
   int cap, num_sfere;
   int contaSfere(lato);
public:
   Cilindro(int = 1);
   Cilindro(const Cilindro&);
   Cilindro& operator=(const Cilindro&);
   int inserisci(lato, int);
   int capacita(int);
   int operator!() const { return num_sfere; }
   ~Cilindro() { delete[] sfere; }
};

int Cilindro::contaSfere(lato t)
{
   int somma_sfere = 0;
   if (t == SINISTRO)
      for (int i = 0; i < cap; i++) {
         if (sfere[i] == 0)
            break;
         somma_sfere++;
      }
   else
      for (int i = cap - 1; i >= 0; i--) {
         if (sfere[i] == 0)
            break;
         somma_sfere++;
      }
   return somma_sfere;
}

Cilindro::Cilindro(int n)
{
   cap = n;
   num_sfere = 0;
   sfere = new int(cap);
   for (int i = 0; i < cap; i++)
      sfere[i] = 0;
}

Cilindro::Cilindro(const Cilindro& s)
{
   cap = s.cap;
   num_sfere = s.num_sfere;
   sfere = new int(cap);
   for (int i = 0; i < cap; i++)
      sfere[i] = s.sfere[i];
}

Cilindro& Cilindro::operator=(const Cilindro& s)
{
   if (this != &s) {
      if (cap != s.cap) {
         delete[] sfere;
         cap = s.cap;
         sfere = new int(cap);
      }
      num_sfere = s.num_sfere;
      for (int i = 0; i < cap; i++)
         sfere[i] = s.sfere[i];
   }
   return *this;
```

```
}

int Cilindro::inserisci(lato t, int v)
{
   int sfera_out = 0;
   if (t == SINISTRO) {
      if (num_sfere == cap)
         sfera_out = sfere[cap - 1];
      for (int i = contaSfere(t); i > 0; i--)
         sfere[i] = sfere[i - 1];
      sfere[0] = v;
   }
   else {
      if (num_sfere == cap)
         sfera_out = sfere[0];
      for (int i = cap - contaSfere(t); i < cap; i++)
         sfere[i - 1] = sfere[i];
      sfere[cap - 1] = v;
   }
   if (sfera_out == 0)
      num_sfere++;
   return sfera_out;
}

int Cilindro::capacita(int p)
{
   if (p < num_sfere)
      return 0;
   int *sfere_nuove = new int(p);
   for (int i = 0; i < p; i++)
      sfere_nuove[i] = 0;
   for (int i = 0; i < contaSfere(SINISTRO); i++)
      sfere_nuove[i] = sfere[i];
   for (int i = 0; i < contaSfere(DESTRO); i++)
      sfere_nuove[p - i - 1] = sfere[cap - i - 1];
   cap = p;
   delete[] sfere;
   sfere = sfere_nuove;
   return 1;
}

ostream& operator<<(ostream& os, const Cilindro& s)
{
   os << '(' << s.cap << ',' << s.num_sfere << ')';
   return os;
}
```