Feladat

Olvassuk be a standard inputról érkező számokat, majd írjuk ki a standard outputra előbb a negatívokat, utána pedig a többit!

Megoldás

A feladat megoldásához egy egész számokat tartalmazó sorozat-típust (tárolót, konténert) készítünk. Egy sorozatot fejelem nélküli kétirányú láncolt listával fogunk ábrázolni. A típusban három pointerrel hivatkozhatunk a listára:

- **`first`**: lista első elemére mutat (üres lista esetén a nil pointer)
- **`last`**: lista utolsó elemére mutat (üres lista esetén a nil pointer)
- **`current`**: bejáráskor a lista aktuális elemére mutat, egyébként nem használjuk

A típushoz az alábbi műveleteket vezetjük be:

Módosító műveletek:

- **`Loext`**: egy egész szám berakása a sorozat elejére
- **`Lopop`**: egy egész szám levétele a sorozat elejéről
- **`Hiext`**: egy egész szám berakása a sorozat végére
- **`Hipop`**: egy egész szám levétele a sorozat végéről

Bejáró műveletek

- **`First`**: a sorozat elejére áll
- **`Next`**: a sorozat következő elemére áll
- **`Eos`**: jelzi, hogy a sorozat végére értünk-e
- **`Current`**: az aktuális értéket adja vissza

A megoldás két program szekvenciája lesz. Az első beolvassa a számokat, és előjelüknek megfelelően a sorozat elejére vagy végére szúrja be őket. A második rész végigjárja a sorozatot, és kiírja az elemeit.

Megoldás C++-ban

Sorozat-típus

Header fájl

A sorozat-típust egy osztállyal valósítjuk meg. Az osztály definícióját a header fájlban (**`sequence.h`**) helyezzük el. Az osztály publikus interfésze a szokásos sorozat-műveleteket tartalmazza.
A **`nil`** a sehova sem mutató cím, amelyet a fájl elején a **`#define nil 0`** sor definiál.
Az **`Exceptions`** felsorolt típus azt az értéket tartalmazza, amelyet a sorozat-osztály hibás felhasználás esetén kivételként dob. Jelen esetben a sorozat akkor dob kivételt, ha egy üres sorozatból ki akarunk venni egy értéket (ld. **`Lopop`** és **`Hipop`** műveletek).

```
#ifndef SEQUENCE_H
#define SEQUENCE_H

#define nil 0

class Sequence{
  public:
    enum Exceptions{EMPTYSEQ};

    Sequence():first(nil),last(nil),current(nil){};
    ~Sequence();

    void Loext(int e);
    int  Lopop();
    void Hiext(int e);
    int  Hipop();

    void First()  {current = first;}
    void Next()   {current = current->next;}
    bool Eos()    const {return current==nil;}
    int  Current()const {return current->cont;}
```

A bejáró műveletek és a konstruktor „inline” definíciót tartalmaz. A konstruktor egy üres sorozatot, azaz egy nulla hosszúságú láncolt listát hoz létre úgy, hogy mindhárom privát adattagot **`nil`**-re állítja. A bejáró műveletek definíciója értelemszerű.

Az osztály privát részében deklaráljuk a copy konstruktort és az értékadás operátort: így letiltjuk a sorozat-objektumok érték szerinti paraméterátadását és az értékadását. Ezt a listaelem-típus definíciója követi. A sorozat reprezentációja a legelső listaelemre mutató **`first`** pointer, a legutolsó listaelemre mutató **`last`** pointer, és az aktuális listaelemre mutató **`current`** pointer.

```
private:
    Sequence(const Sequence&);
    Sequence& operator=(const Sequence&);

    struct Node{
        int   cont;
        Node* next;
        Node* prev;
        Node(int c, Node* n, Node* p):
                        val(c), next(n), prev(p){};
    };

    Node* first;
    Node* last;
    Node* current;
};

#endif
```

Implementációs fájl

Az osztály implementációs fájlja (**sequence.cpp**) tartalmazza a műveletek megvalósítását. A destruktor felszabadítja a sorozatot ábrázoló listát, hiszen az dinamikusan lett létrehozva.

```
#include ”sequence.h”

Sequence::~Sequence()
{
    Node *p, *q;
    q = first;
    while( q!=nil){
        p = q;
        q = q->next;
        delete p;
    }
}
```

A **Loext** művelet létrehoz egy új listaelemet, megfelelően kitölti, és befűzi azt a lista elejére. Ha a lista eredetileg üres volt, akkor az utolsó elemre mutató **last** pointert is be kell állítani.

```
void Sequence::Loext(int e)
{
    Node* p = new Node(e,first,nil);
    if(first!=nil){
        first->prev = p;
    }
    first = p;
    if(last==nil){
        last = p;
    }
}
```

A **Lopop** művelet először ellenőrzi, hogy van-e elem a listában. Ha nincs, akkor **EMPTYSEQ** kivételt dob, egyébként kifűzi a lista legelső elemét, a benne tárolt értéket elmenti, és a listaelemet felszabadítja. Ha a lista eredetileg egyelemű volt, akkor kifűzés helyett az első és utolsó elemre mutató pointereket kell **nil**-re állítani.

```
int  Sequence::Lopop()
{
    if(first==nil) throw EMPTYSEQ;
    int e = first->val;
    Node* p = first;
    first = first->next;
    delete p;
    if(first!=nil) first->prev = nil;
    else           last = nil;

    return e;
}
```

A **Hiext** és **Hipop** műveletek megvalósítása analóg a **Loext** és **Lopop** műveletekével.

```
void Sequence::Hiext(int e)
{
    Node* p = new Node(e,nil,last);
    if(last!=nil){
        last->next = p;
    }
    last = p;
    if(first==nil){
        first = p;
    }
}

int  Sequence::Hipop()
{
    if(last==nil)throw EMPTYSEQ;
    int e = last->val;
    Node* p = last;
    last = last->prev;
    delete p;
    if(last!=nil) last->next = nil;
    else          first = nil;

  return e;
}
```

A főprogram

A főprogramban létrehozunk egy üres sorozatot-objektumot, majd a standard inputról érkező számokat előjelüktől függően belerakjuk a sorozatba. A beolvasás után bejárjuk a sorozatot, és kiírjuk az elemeit a standard outputra.

```
#include <iostream>
#include "sequence.h"

using namespace std;

int main()
{
    Sequence x;
    int i;
    while(cin>>i){
        if (i>0) x.Hiext(i);
        else     x.Loext(i);
    }

    for(x.First(); !x.Eos(); x.Next()){
        cout<<x.Current()<<endl;
    }

  return 0;
}
```

Felsorolt típusú kivétel kezelése

A felsorolt (nem beágyazott osztály) típusú kivételek csak a hiba jelzésére alkalmasak, nem képesek más információt a hibát kiváltó hívás helyére eljuttatni.
Ha az első feladat főprogramjában egy sorozat-objektumot hibásan használunk, akkor ez egy olyan kivételt vált ki, amelynek típusa a sorozat-osztályon belül van definiálva. Ezért erre a típusra a főprogramban csak minősítve hivatkozhatunk. Mivel a felsorolt típusnak elvileg több értéke is lehet, a kivételkezelésben meg kell vizsgálni a dobott kivétel-objektum értékét.

```
Sequence x;

try{
    cout << x.Lopop() << endl;
}catch(Sequence::Exceptions e){
    if(e==Sequence::EMPTYSEQ){
        cout<<"üres sorozatból nem lehet elemet törölni"<<endl;
    }
}
```

Egy újabb feladat

Olvassuk be a standard inputról érkező számokat, majd írjuk ki őket az érkezésük sorrendjében úgy, hogy megadjuk minden szám minden előfordulásánál a szám összes előfordulásának számát!

Megoldás

A feladat megoldásához egyidejűleg két bejáró kell: az egyikkel végig megyünk az elemeken, - a másikkal minden elemre megszámoljuk annak előfordulásait. Egy bejáró-típust készítünk a sorozat-típushoz, annak beágyazott osztályaként.

Megoldás C++-ban

Sorozat-típus bejáró-típussal

Header fájl

A sorozat-típus osztály-definíciójából kivesszük a bejárásra vonatkozó elemeket (a **`current`** adattagot, valamint a **`First`**, a **`Next`**, az **`Eos`**, a **`Current`** metódusokat), és a beágyazott bejáró-típusba helyezzük el őket. A beágyazott típus a sorozat-típus publikus részébe kerül, hogy szolgáltatásait kívülről el tudjuk érni, és kölcsönösen barátként jelölik meg egymást.

```
#ifndef SEQUENCE_H
#define SEQUENCE_H

#define nil 0

class Sequence{
  public:
    enum Exceptions{EMPTYSEQ};

    Sequence():first(nil),last(nil){};
    ~Sequence();
    Sequence(const Sequence& s);
    Sequence& operator=(const Sequence& s);

    void Loext(int e);
    int  Lopop();
    void Hiext(int e);
    int  Hipop();

private:
    struct Node{
        int   cont;
        Node* next;
        Node* prev;
        Node(int c,Node* n,Node* p):cont(c),next(n),prev(p){};
    };

    Node* first;
    Node* last;

  public:
    friend class Iterator;
    class Iterator{
        friend class Sequence;
      public:
        Iterator(Sequence& s):seq(&s),current(nil){};

        void First()  {current = seq->first;}
        void Next()   {current = current->next;}
        bool Eos()    const {return current==nil;}
        int  Current()const {return current->cont;}

      private:
        Sequence *seq;
        Node* current;
    };
};

#endif
```

Egy bejáró-objektum ismeri egy sorozat-objektumnak a címét, és ennek a sorozatnak egyik listaelemére mutató **`current`** pointert.

Az implementációs fájl nem változik, mert eddig is csak a sorozat módosító műveleteinek definícióját tartalmazta, és a bejáróra vonatkozó műveletek mind „inline” metódusként lettek megvalósítva.

A főprogram

A főprogramban létrehozunk két bejáró-objektumot: az egyikkel végigmegyünk a lista elemein, a másikat pedig a számlálás megvalósítására használjuk.

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

using namespace std;

int main()
{
    Sequence x;
    int i;
    while( cin>>i){
        x.Hiext(i);
    }

    Sequence::Iterator it1(x);
    Sequence::Iterator it2(x);
    for(it1.First(); !it1.Eos(); it1.Next()){
        i = it1.Current();
        int s = 0;
        for(it2.First(); !it2.Eos(); it2.Next()){
            if (it2.Current()==i){
                s++;
            }
        }
        cout << i << " előfordulásainak száma: " << s << endl;
    }

  return 0;
}
```

Elem törlése bejárás közben

Egy problémával még foglalkoznunk kell: sem az eredeti típus, sem a bejáróval kiegészített nem viselkedik jól, ha bejárás alatt a sorozatból törlünk egy olyan elemet, amelyre egy **`current`** pointer hivatkozik. A probléma elkerülésére több megoldás is elképzelhető:

- Teljes kizárás: A törlő műveletek kivételt dobnak bejárás esetén.
- Elemszintű kizárás: A törlő műveletek kivételt dobnak, ha olyan elemre vonatkoznak, amelyre bejáró hivatkozik.
- Törlés késleltetés: A törlendő elem csak akkor törlődik, ha már nem hivatkozik rá bejáró.

A fenti lehetőségek közül az egyszerűség kedvéért a legelsőt fogjuk megvalósítani. Ehhez felveszünk egy új kivétel értéket (**`UNDERTRAVERSAL`**) és egy bejáró-számlálót (**`iteratorCount`**) a sorozat-osztályban. A számlálót a bejáró konstruktora növeli, destruktora csökkenti.

```
#ifndef SEQUENCE_H
#define SEQUENCE_H

#define nil 0

class Sequence{
  public:
    enum Exceptions{EMPTYSEQ, UNDERTRAVERSAL};

    Sequence():first(nil),last(nil){};
    ~Sequence();
    Sequence(const Sequence& s);
    Sequence& operator=(const Sequence& s);

    void Loext(int e);
    int  Lopop();
    void Hiext(int e);
    int  Hipop();

private:
    struct Node{
        int   cont;
        Node* next;
        Node* prev;
        Node(int c,Node* n,Node* p):cont(c),next(n),prev(p){};
    };

    Node* first;
    Node* last;
    int   iteratorCount;

  public:
    friend class Iterator;
    class Iterator{
        friend class Sequence;
      public:
        Iterator(Sequence& s):seq(&s),current(nil)
                    {seq->iteratorCount++;}
        ~Iterator() {seq->iteratorCount--;}

        void First()  {current = seq->first;}
        void Next()   {current = current->next;}
        bool Eos()    const {return current==nil;}
        int  Current()const {return current->cont;}

      private:
        Sequence *seq;
        Node* current;
    };
};

#endif
```

A sorozat törlő műveletei annyiban változnak, hogy a művelet elvégzése előtt ellenőrzik a számláló értékét, ha az nem nulla, akkor kivételt dobnak.

```
...

int  Sequence::Lopop()
{
    if(iteratorCount!=0) throw UNDERTRAVERSAL;
    if(first==nil) throw EMPTYSEQ;

    int e = first->val;
    Node* p = first;
    first = first->next;
    delete p;
    if(first!=nil) first->prev = nil;
    else           last = nil;

  return e;
}

...

int  Sequence::Hipop()
{
    if(iteratorCount!=0) throw UNDERTRAVERSAL;
    if(last==nil)throw EMPTYSEQ;

    int e = last->val;
    Node* p = last;
    last = last->prev;
    delete p;
    if(last!=nil) last->next = nil;
    else          first = nil;

  return e;
```