Feladat

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

Absztrakt megoldás

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

Sorozat típus

A feladat megoldásához definiálni kell egy egész számokat tartalmazó sorozat típust.

Típusérték-halmaz

Egész számok sorozatai: Z^*. (Közöttük van az üres sorozat is.)

Típus-műveletek

A típushoz az alábbi műveleteket vezetjük be:

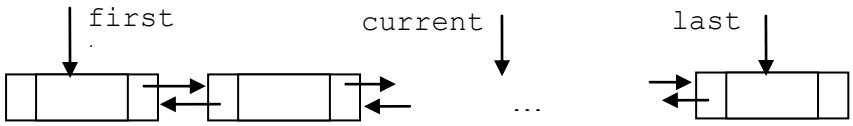
Módosító műveletek

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

Bejáró műveletek

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

Reprezentáció



Egy sorozatot fejelem nélküli kétirányú láncolt listával fogunk ábrázolni. Egy listaelem mezői: **`prev`** (pointer), **`val`** (érték), **`next`** (pointer). A láncolt listát az alábbi pointerek azonosítják:

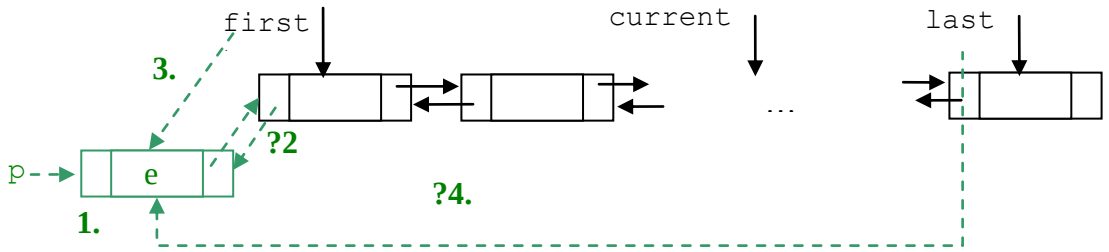
- **`first`**: lista első elemére mutat (üres lista esetén a null pointer)
- **`last`**: lista utolsó elemére mutat (üres lista esetén a null pointer)
- **`current`**: bejáráskor a lista aktuális elemére mutat, egyébként nem használjuk

Implementáció

A bejáró műveletek a `current` pointerrel kapcsolatos egyszerű értékadásokkal implementálhatóak. Itt nem térünk ki itt a részletezésükre, a C++ implementációban megtaláljuk őket.

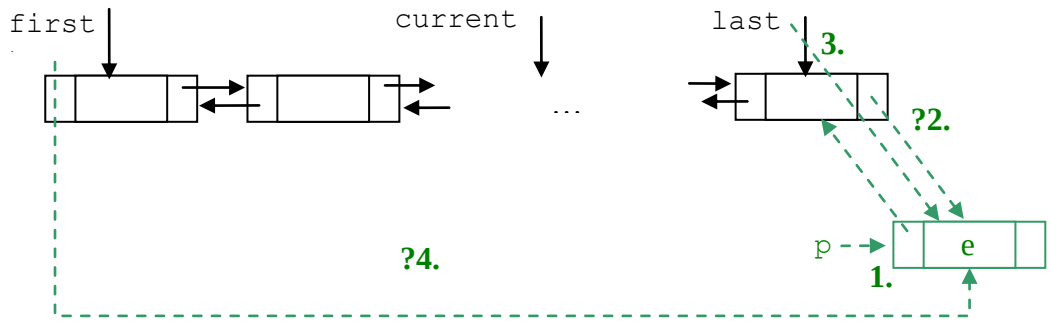
A módosító műveletek hatását legegyszerűbben egy-egy ábrán mutathatjuk be. Megfigyelhető az is, hogy a *Loext* és *Hiext*, illetve a *Lopop* és *Hipop* műveletpárok mennyire hasonlóak, egymás ellentétpárjai.

1. *Loext*



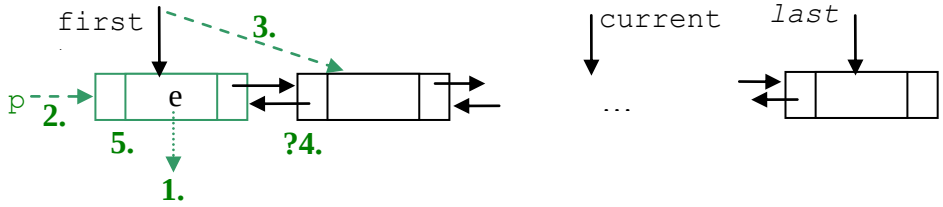
1. új listaelem létrehozása egy `p` segédpointerrel, és annak kitöltése a `prev`, `val`, `next` sorrendjében: `nil`, `e`, `first` mezőértékekkel
2. ha a lista nem volt üres, akkor az eddigi első listaelem `prev` mezője az új listaelemre mutasson
3. `a first` pointer az új listaelemre mutasson
4. ha eddig üres volt a lista, akkor a `last` pointer az új listaelemre mutasson

2. Hiext



1. új listaelem létrehozása egy `p` segédpointerrel, és annak kitöltése a `prev`, `val`, `next` sorrendjében: `last`, `e`, `nil` mezőértékekkel
2. ha a lista nem volt üres, akkor az eddigi utolsó listaelem `next` mezője az új listaelemre mutasson
3. `a last` pointer az új listaelemre mutasson
4. ha eddig üres volt a lista, akkor a `first` pointer az új listaelemre mutasson

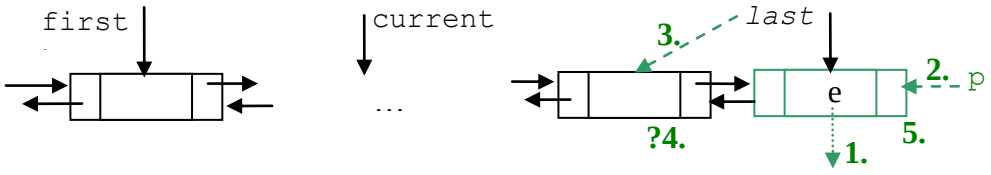
3. Lopop



Üres lista esetén hibás művelet, egyébként

1. első listaelem értékének kiolvasása
2. a `p` segédpointer mutasson az első listaelemre
3. `a first` pointer az első listaelem `next` pointerének értékét kapja
4. ha van második listaelem, akkor annak `prev` pointere legyen `nil`, különben a `last` pointer legyen `nil`.
5. szabadítsuk fel az első listaelemet

4. Hipop



Üres lista esetén hibás művelet, egyébként

1. utolsó listaelem értékének kiolvasása
2. a `p` segédpointer mutasson az utolsó listaelemre
3. `a last` pointer az utolsó listaelem `prev` pointerének értékét kapja
4. ha van utolsó előtti listaelem, akkor annak `next` pointere legyen `nil`, különben a `first` pointer legyen `nil`.
5. szabadítsuk fel az utolsó listaelemet

Megoldás C++-ban

Sorozat-típus

A sorozat-típust egy osztállyal valósítjuk meg. Az osztály deklarációját a `sequence.h` fejállományban helyezzük el, a metódusok implementációit a `sequence.cpp` forrásállományban, amely természetesen „inklúdolja” a felállományt.

Az osztály publikus része a típusspecifikációban felsorolt műveleteken kívül a konstruktort, a destruktort, a másoló konstruktort és az értékadás operátort tartalmazza.

Az `Exceptions` felsorolt típus azt a hiba-objektumot tartalmazza, amelyet a sorozat-osztály a `Lopop` és `Hipop` műveletekben dob, ha egy üres sorozatból ki akarunk venni egy értéket.

A bejáró műveletek és a konstruktor „inline” definíciót tartalmaznak. A `Next` műveletre kétféle ++ operátort is adunk: az első utótagként használható, a második előtagként. (Vegyük észre, hogy az első operátornál szükség van a másoló konstruktorra.)

```
#ifndef SEQUENCE_H
#define SEQUENCE_H

class Sequence{
  public:
    enum Exceptions{EMPTYSEQ};

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

    void Loext(int e);
    int Lopop();
    void Hiext(int e);
    int Hipop();
    int Current() const {return current->val;}
    void First()   {current = first;}
    bool End()     const {return current==NULL;}
    void Next()    {current = current->next;}
    Sequence operator++(int){
        Sequence s = *this;
        current = current->next;
        return s;
    }
    Sequence& operator++(){
       current = current->next;
        return *this;
    }
```

A privát részbe a listaelem-típus definíciója kerül. A sorozat reprezentációja a legelső listaelemre mutató `first` pointer, a legutolsó listaelemre mutató `last` pointer, és az aktuális listaelemre mutató `current` pointer.

```
  private:
    struct Node{
        int val;
        Node* next;
        Node* prev;
        Node(int c, Node* n, Node* p):
                        val(c), next(n), prev(p){};
    };
    Node* first;
    Node* last;
    Node* current;
};
```

Az osztály definícióját a műveletek implementációi követik.

1. Konstruktor

Tevékenység: A konstruktor egy üres sorozatot, azaz egy nulla hosszúságú láncolt listát hoz létre úgy, hogy mindhárom privát adattagot (`first`, `last`, `current`) `NULL`-re állítja.

Bemenő adatok: -
Kimenő adatok: új üres sorozat (`Sequence`)
Definíció: (inline) `Sequence():first(NULL),last(NULL),current(NULL){}`

2. Destruktor

Tevékenység: Megszűntet egy sorozatot, felszabadítja a listaelemeit. Ehhez a `p` és `q` pointert vezeti végig a sorozaton úgy, hogy a `q` mindig eggyel előbbre mutat, mint a `p`.

Bemenő adatok: alapértelmezett sorozat (`Sequence`)
Kimenő adatok: -
Definíció: (inline)

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

3. Másoló konstruktor

Tevékenység: A konstruktor létrehoz egy új sorozatot, amely pontosan megegyezik a paraméterként megadott sorozattal. A másoló konstruktor a `p` pointert vezeti végig a másolandó sorozaton, és a `q` pointert használja az új sorozat felépítéséhez.
Bemenő adatok: egy sorozat (`s:Sequence`)
Kimenő adatok: új sorozat (`Sequence`)

Definíció:

```
Sequence::Sequence(const Sequence& s){
   if(s.first==NULL){
      first = last = NULL;
   }else{
      Node* q = new Node(s.first->val,NULL,NULL);
      first = q;
      for(Node* p=s.first->next;p!=NULL;p=p->next){
         q = new Node(p->val,NULL,q);
         q->prev->next = q;
      }
      last = q;
   }
   current = first;
   while(current!=NULL &&
                  current->val!=b.current->val){
       current=current->next;
   }
}
```

4. Értékadás operátor

Tevékenység: Az operátor értékül adja az értékadás jobboldalán adott sorozatot a baloldalán álló sorozatnak. Ennek eredményként a baloldali sorozat ugyanannyi, ugyanolyan értékű és ugyanabban a sorrendben elhelyezett elemet fog tartalmazni. Az értékadás operátor felszabadítja az alap objektumot, ha az különbözik másolandó objektumtól (lásd destruktor), majd létrehozza az új sorozatot (lásd másoló konstruktor).

Bemenő adatok: baloldali sorozat (`this:Sequence`)
a jobboldali sorozat (`s:Sequence`)

Kimenő adatok: baloldali sorozat (`this:Sequence`)

Definíció:

```
Sequence& Sequence::operator=(const Sequence& s){
   if(&s==this) return *this;
   Node* p = first;
   while(p!=NULL){
      Node* q = p->next;
      delete p;
      p = q;
   }
   if(s.first==NULL){
      first = last = NULL;
   }else{
      Node* q = new Node(s.first->val,NULL,NULL);
      first = q;
      for(Node* p=s.first->next;p!=NULL;p=p->next){
         q = new Node(p->val,NULL,q);
         q->prev->next = q;
      }
      last = q;
   }
   current = first;
   while(current!=NULL &&
                  current->val!=b.current->val){
      current=current->next;
```

```
    }
    return *this;
}
```

5. Current

Tevékenység: A current pointer által mutatott sorozatbeli listaelem értékét adja vissza.

Bemenő adatok: alapértelmezett sorozat (`Sequence`)
Kimenő adatok: érték (**`int`**)
Definíció: (inline) `int Current() const {return current->val;}`

6. First

Tevékenység: A `current` pointert ráállítja a sorozat első elemére, amit a `first` pointer mutat. A sorozat bejárásánál ez a kezdeti értékadás.

Bemenő adatok: alapértelmezett sorozat (`Sequence`)
Kimenő adatok: alapértelmezett sorozat (`Sequence`)
Definíció: (inline) `void First()  {current = first;}`

7. End

Tevékenység: Igaz értéket ad vissza, ha a `current` pointer „lefut” a sorozatról, azaz NULL értéket tartalmaz. A sorozat bejárásánál ez a terminálási feltétel.

Bemenő adatok: alapértelmezett sorozat (`Sequence`)
Kimenő adatok: logikai érték (**`bool`**)
Definíció: (inline) `bool End() const {return current==NULL;}`

8. Next

Tevékenység: Továbblépteti a `current` pointert a sorozat következő elemére. A sorozat bejárásánál ez a ciklusmag iterációs lépése.

Bemenő adatok: alapértelmezett sorozat (`Sequence`)
Kimenő adatok: alapértelmezett sorozat (`Sequence`)
Definíció: (inline) `void Next(){current = current->next;}`
Alternatív definíciók: (inline)

```
Sequence operator++(int){
    Sequence s = *this;
    current = current->next;
    return s;
}

Sequence& operator++(){
    current = current->next;
    return *this;
}
```

9. Loext

Tevékenység: A `Loext` művelet egy új, kitöltött listaelemet fűz be a lista elejére, az absztrakt implementációnak megfelelően

Bemenő adatok: alapértelmezett sorozat (`Sequence`)
új érték (`e:int`)
Kimenő adatok: alapértelmezett sorozat (`Sequence`)
Definíció:

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

10. Hiext

Tevékenység: A `Hiext` művelet egy új, kitöltött listaelemet fűz be a lista végére, az absztrakt implementációnak megfelelően

Bemenő adatok: alapértelmezett sorozat (`Sequence`)
új érték (`e:int`)
Kimenő adatok: alapértelmezett sorozat (`Sequence`)
Definíció:

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

11. Lopop

Tevékenység: A `Lopop` művelet megszűnteti a lista legelső elemét miután a benne tárolt értéket elmentette feltéve, hogy a lista nem üres (különben `EMPTYSEQ` kivételt dob), az absztrakt implementációnak megfelelően

Bemenő adatok: alapértelmezett sorozat (`Sequence`)
Kimenő adatok: alapértelmezett sorozat (`Sequence`)
új érték (`e : int`)
Definíció:

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

12. Hipop

Tevékenység: A `Lopop` művelet megszűnteti a lista legelső elemét miután a benne tárolt értéket elmentette feltéve, hogy a lista nem üres (különben `EMPTYSEQ` kivételt dob), az absztrakt implementációnak megfelelően

Bemenő adatok: alapértelmezett sorozat (`Sequence`)
Kimenő adatok: alapértelmezett sorozat (`Sequence`)
új érték (`e : int`)
Definíció:

```
int Sequence::Hipop(){
   if(last==NULL)throw EMPTYSEQ;
   Item e = last->val;
   Node* p = last;
   last = last->prev;
   delete p;
   if(last!=NULL) last->next = NULL;
   else           first = NULL;
   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;
    cin >> i;
    while(i!=0){
        if (i>0) x.Hiext(i);
        else     x.Loext(i);
        cin >> i;
    }

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

  return 0;
}
```

Tesztelési terv

Főprogram tesztelése (fekete és fehérdoboz tesztelés)

1. Nulla darab szám esete.
2. Csupa pozitív szám esete.
3. Csupa negatív szám esete.
4. Egyetlen negatív szám (elején, végén, középen) a pozitívok között.
5. Egyetlen pozítív szám (elején, végén, középen) a negatívok között.
6. Általános adatsor.

Osztály-sablon tesztelése (fekete és fehérdoboz tesztelés)

Ehhez célszerű egy külön tesztprogramot készíteni, amely az összes típusműveletet kipróbálja. Ilyenkor minden típusműveletre külön-külön tesztelési tervet kell készíteni.

Például a Lopop tesztelése:
1. Üres sorozat esete.
2. Egy elemű sorozat esete.
3. Több elemű sorozat esete.

Fejlesztési lehetőségek[1]

Egyik fejlesztési lehetőség, hogy a sorozat-típus osztály-definíciójából kivesszük a bejárásra vonatkozó elemeket (a `current` adattagot, a `First`, a `Next`, `operator++`, az `End`, a `Current` metódusokat, valamint a `current` adattag beállítását a konstruktorokból és az értékadás operátorból), és ezeket a beágyazott bejáró-típusba helyezzük el őket. A beágyazott típus a sorozat-típus publikus részébe kerül, és azért hogy hozzáférjen a sorozat-típus privát részeihez a sorozat-típus barátjának deklaráljuk. Egy bejáró-objektum ismeri egy sorozat-objektumnak a címét, és nyilvántart ennek a sorozatnak egyik listaelemére mutató `current` pointert.

Másik probléma, hogy sem az eredeti típus, sem az előbb említett bejáróval kiegészített típus 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, amennyiben van bejárás a sorozatra. Ehhez elég a sorozat-objektumban nyilvántartani a bejárók számát.
- Elemszintű kizárás: A törlő műveletek kivételt dobnak, ha olyan elemre vonatkoznak, amelyre valamelyik bejáró hivatkozik. Ehhez nyilván kell tartani a sorozat-objektumban a bejárókat.
- Törlés késleltetés: A törlendő elem csak akkor törlődik, ha már nem hivatkozik rá bejáró. Ehhez azon kívül, hogy nyilvántartjuk a sorozat-objektum bejáróit, nyilván kell tartani a törölt listaelemeket, amelyeket megfelelő pillanatban ki is kell törölni.

[1] Lásd előadás anyagát