Feladat

Készítsen egy zsák típust! Alkalmazzon osztályt! A zsákokat rendezett láncolt listával ábrázolja! Implementálja a szokásos műveleteket, egészítse ki az osztályt a kényelmes és biztonságos használat érdekében megfelelő metódusokkal (zsákbeolvasó operátor», zsákkiíró operátor«), alkalmazzon kivételkezelést, készítsen teszt környezetet, és bontsa modulokra a programját! A teszt környezet hívjon meg egy olyan barát-operátort is, amely kiszámítja két zsák unióját (a közös elemek előfordulása összegződik)! Az unióképzés műveletigénye: $O(m+n)$, ahol m és n a két zsáknak megfelelő halmazok elemszáma.

Zsák

Típusspecifikáció

A zsák olyan halmaz, amelyben az elemeknek multiplicitása van, tehát a halmaz egyfajta általánosításaként fogható fel: egy halmazban az $\in$ operátor $\mathbb{L}$-be képez, de definiálható pl. a következő operátor:

$$\begin{aligned} mul &: \mathbb{S} \times T \to \mathbb{N} \\ mul(S,t) &:= \begin{cases} 1 & \text{ha } t \in S \\ 0 & \text{egyébként} \end{cases} \end{aligned}$$

Zsákban pedig $mul(B,t)$ tetszőleges nemnegatív értéket felvehet.

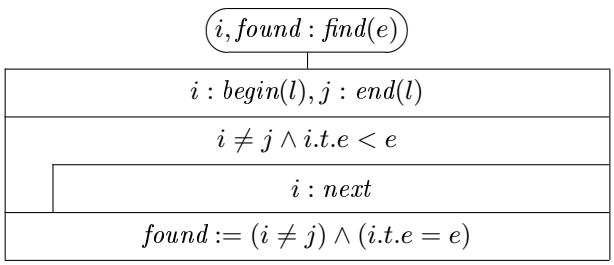
Reprezentáció

Itt egyszerű a dolgunk: a feladatkiírásban szerepel, hogy rendezett, láncolt listát kell használnunk. A lista elemei *(elem, multiplicitás)* =: (e,m) párok. A rendezettséget a hozzáadó és az eltávolító műveletek tartják fent. Ha egy elem multiplicitása nullára csökken, akkor az elemhez tartozó lista-elemet kitöröljük a listából. Új elem beszúrásakor megkeressük a listában az elemhez tartozó helyet, és ha már létezik a listában megfelelő elem, akkor növeljük annak a multiplicitását, ha pedig nem, akkor beszúrunk egy új elemet.

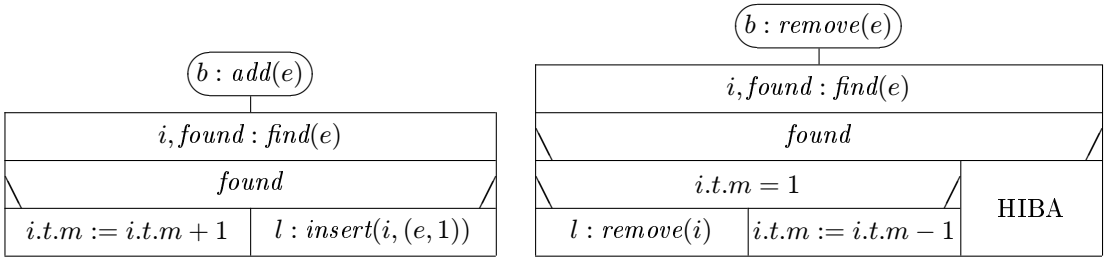
Absztrakt implementáció

Keresés

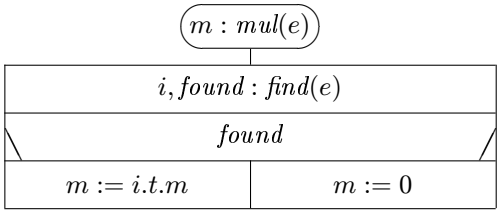
Ezt az eljárást a publikus műveletek lenti megvalósításánál használjuk.



Hozzáadás, Törlés

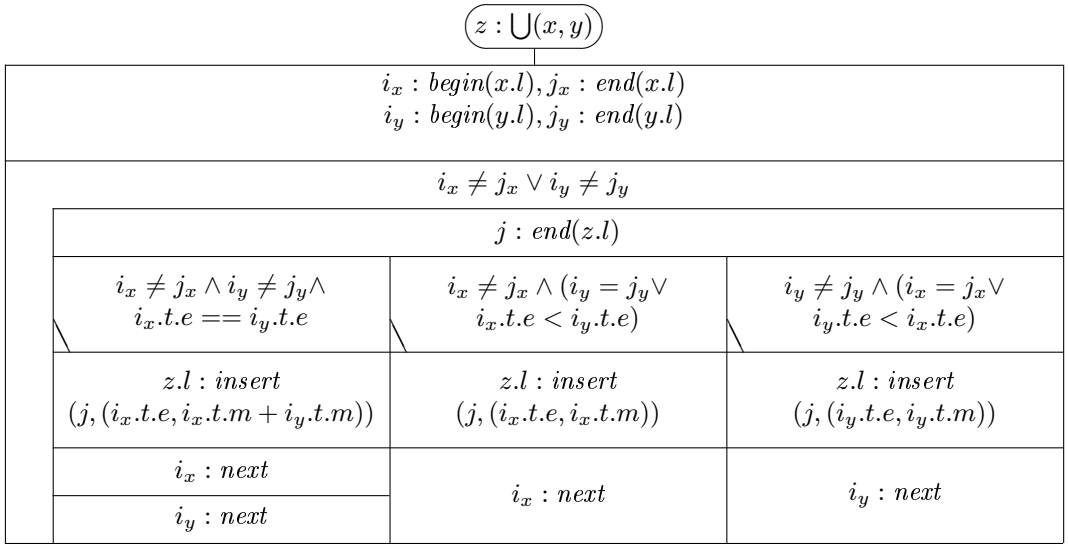


Elem multiplicitásának lekérdezése



Zsákok uniója

Természetesen elegendő lenne az eddigi *insert* művelet is két zsák uniójának kiszámolásához, azonban ez egyrészt minden elemet annyiszor dolgozna fel, amennyi a multiplicitása, másrészt nem használná ki azt, hogy a két forrás-zsákot reprezentáló lista maga is már rendezve van. Ezért az alábbi, az elemenkénti feldolgozásra épülő, $O(n_1 + n_2)$ műveletigényű unió-műveletet használjuk:



Fekete doboz-tesztelés

1. Üres zsákba elem berakása
2. Új elem berakása zsákba
3. Bentlévő elem újbóli berakása zsákba
4. Egynél többször szereplő elem törlése
5. Egyszer szereplő elem törlése

6. Zsákban nem lévő elem törlési kísérlete
7. Üres zsákok uniója
8. Üres és nemüres zsák uniója
9. Diszjunkt zsákok uniója
10. Nem-diszjunkt zsákok uniója

C++ implementáció

A zsák típust természetesen mint osztály-sablon implementáljuk, a sablon típusparamétere az elemek típusa, és ezek rendezése. Az elemek eltárolására egy megfelelő elem-típusú láncolt listát használunk.

```
template<typename T, typename Compare=std::less<T> >
class Bag
{
public:
    typedef T elem_t;
    typedef Compare compare_t;

private:
    struct listelem_t
    {
        elem_t elem;
        int    m;

        listelem_t (const elem_t &elem_, int m_ = 1):
            elem (elem_), m (m_) {}
    };

    typedef List<listelem_t> list_t;
    typedef typename list_t::iterator list_iterator;
    typedef typename list_t::const_iterator const_list_iterator;
    list_t list;

    // ...

};
```

Konstruktor, destruktor

Default konstruktorra tulajdonképpen nincs szükség: a láncolt lista alaphelyzetben üresen indul, ami pont megfelel az üres zsáknak.

```
Bag (){};
```

Értékadás, másolás

Másoló konstruktort és értékadó operátort is készítünk osztályunkhoz, természetesen mindkét művelet egyszerűen az *(elem, multiplicitás)* párok listájának átmásolását jelenti.

```
// Copying, assignment
Bag (const Bag &other) : list (other.list) {}

Bag& operator= (const Bag &other) {
    if (this == &other) return *this;

    list = other.list;
    return *this;
```

Iterálás

Jogos igény, hogy zsákunk kövesse az STL konténerek interfész-mintáit. Ezért a zsák elemeit `iterator` és `const_iterator` típusú iterátorokkal járhatjuk be. Bejáráskor minden egyes elemet

a multiplicitásának megfelelőször kapjuk. Ezt úgy implementáljuk, hogy az iterátorban bennevan az is, hogy ő éppen hányadik példányt jelenti:

```
class iterator
{
    list_iterator iter;
    int i;

    iterator (const list_iterator &iter_) : iter (iter_), i (iter ? iter->m : 0) {}
    friend class Bag;

public:
    typedef elem_t& reference;
    typedef elem_t* pointer;

    reference operator* ()  { return iter->elem; }
    pointer   operator-> () { return iter->elem; }

    iterator&       operator++()    { next (); return *this; }
    const iterator operator++(int) { iterator ret = *this; ++*this; return ret; }

    bool operator== (const iterator &other) const { return other.iter == iter && other.i =
    bool operator!= (const iterator &other) const { return !(other == *this); }

private:
    void next () {
        if (--i) return;

        ++iter;
        i = iter ? iter->m : 0;
    }
};

// const_iterator hasonlóan

iterator       begin ()       { return iterator (list.begin ()); }
const_iterator begin () const { return const_iterator (list.begin ()); }

iterator       end ()       { return iterator (list.end ()); }
const_iterator end () const { return const_iterator (list.end ()); }
```

Hozzáadás, elvétel, multiplicitás, unió

A tárgyalt absztrakt programok egyszerű implementációi.

```
template<typename T, typename Compare>
void Bag<T, Compare>::add (const elem_t &elem)
{
    list_iterator i;

    if (find (elem, i))
    {
        ++i->m;
    } else {
        list.insert (i, listelem_t (elem, 1));
    }
}
```

```
template<typename T, typename Compare>
void Bag<T, Compare>::remove (const elem_t &elem)
{
    list_iterator i;

    if (find (elem, i))
    {
        if (!--i->m)
            list.remove (i);
    } else {
            throw std::runtime_error ("Trying to remove element not in bag");
    }


template<typename T, typename Compare>
int Bag<T, Compare>::operator[] (const elem_t &elem) const
{
    const_list_iterator i;
    if (find (elem, i))
        return i->m;
    else
        return 0;


template<typename T, typename Compare>
Bag<T, Compare> operator+ (const Bag<T, Compare> &lhs,
                           const Bag<T, Compare> &rhs)
{
    typedef Bag<T, Compare> bag_t;
    typedef typename bag_t::list_t list_t;
    typedef typename bag_t::listelem_t listelem_t;
    typedef typename bag_t::compare_t compare_t;

    bag_t ret;

    typename list_t::const_iterator i = lhs.list.begin ();
    typename list_t::const_iterator j = rhs.list.begin ();

    while (i != lhs.list.end () || j != rhs.list.end ())
    {
        if (i != lhs.list.end () && j != rhs.list.end () && i->elem == j->elem)
        {
            ret.list.insert (ret.list.end (), listelem_t (i->elem, i->m + j->m));
            ++i;
            ++j;
        } else if (i != lhs.list.end () && (j == rhs.list.end () || compare_t()(i->elem, j->ele
            ret.list.insert (ret.list.end (), listelem_t (i->elem, i->m));
            ++i;
        } else if (j != rhs.list.end () && (i == lhs.list.end () || compare_t()(j->elem, i->ele
            ret.list.insert (ret.list.end (), listelem_t (j->elem, j->m));
            ++j;
        }
    }

    return ret;
}
```

Beolvasás/kiírás

A zsák tartalmát úgy szerializáljuk, hogy kiírjuk a lista elemeinek számát, majd egymás után a lista elemeit *multiplicitás␣elem* formátumban:

```
template<typename T, typename Compare>
std::ostream& operator<< (std::ostream &str, const Bag<T, Compare> &bag)
{
    typedef Bag<T, Compare> bag_t;

    str << bag.list.size () << '␣';
    for (typename bag_t::list_t::const_iterator i = bag.list.begin (); i != bag.list.end (); +
        str << i->m << '␣' << i->elem << '␣';

    return str;
}

template<typename T, typename Compare>
std::istream& operator>> (std::istream &str, Bag<T, Compare> &bag)
{
    typedef Bag<T, Compare> bag_t;

    bag.clear ();

    size_t s;
    for (str >> s; str && s; --s)
    {
        int m;
        typename bag_t::elem_t elem;
        str >> m >> elem;

        if (str)
            bag.list.insert (bag.list.end (), typename bag_t::listelem_t (elem, m));
    }

    return str;
}
```

Tesztelési terv

1. (l. absztrakt implementációs rész)
2. Zsák kiírása stream-be
3. Zsák beolvasása stream-ből

Függelék: Láncolt lista

Típusspecifikáció

Típusérték-halmazunk $L = T^*$, vagyis a T elem-típusú sorozatok.

Műveleteink a beszúrás, a törlés, és a végiglépdelés. Ezek formális leírásához szükségünk lenne az L_T *feletti iterátor* típusra, annak műveleteire, stb. Ezektől itt eltekintünk. A feladat kiírásában szereplő „rendezett láncolt lista" ebből az egyszerű, láncolt listából jön létre olymódon, hogy a zsák típus a reprezentációjául szolgáló listát speciális módon kezeli.

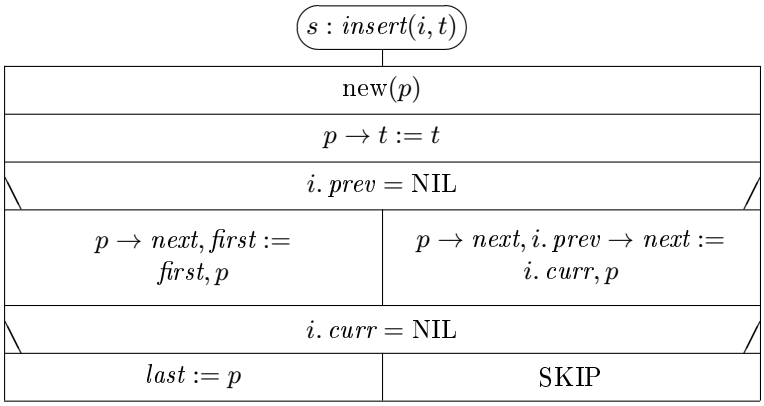
Reprezentáció

A láncolt listát (minő meglepő) láncoltan reprezentáljuk, a kezdő- és a vég-csomópont eltárolásával, az iterátorok pedig egy (előző, aktuális) mutató-párt tartalmaznak.

Absztrakt implementáció

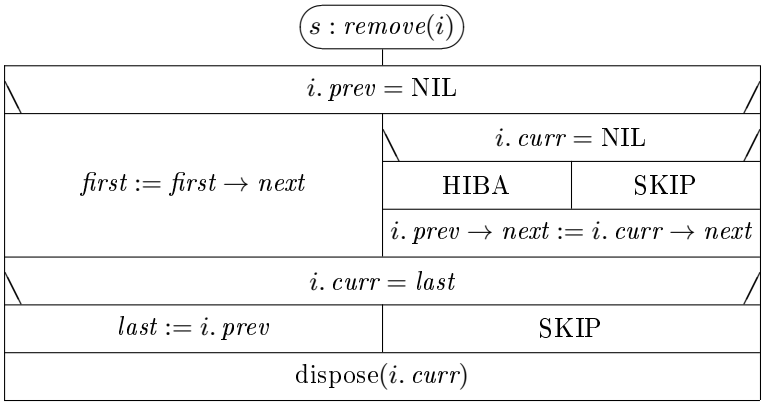
Beszúrás

Beszúráskor az $i.prev = \text{NIL}$ esettel reprezentált legelőre-beszúrás speciálisan kezelendő (mivel a feladat kiírása nem tartalmazta, hogy fejelemes listát használjunk).



Törlés

Töröléskor először kiláncoljuk az iterátor által mutatott elemet, majd elvégezzük a felszabadítást, végül, ha az utolsó vagy az első elemet töröltük, akkor értelemszerűen frissítjük az utolsó/első mutatót:



Fekete doboz-tesztelés

1. Üres listába beszúrás
2. Beszúrás nem üres lista elejére, végére, közepére
3. Egyelemű listából elem törlése
4. Nemlétező elem törlése