Duba Emese Mária

AI83KT

Programozás I. Beadandó

Ezen dokumentumot Duba Emese Mária ELTE IK Programtervező Informatikus szakjára járó hallgató készítette Programozás I. tárgy első beadandó feladat dokumentációjaként.

Követelmény

Csak a funkcionálisan működő (a bemutatott programnak a kitűzött feladatot kell megoldania), az előírt kiegészítő követelményeknek megfelelő és elektronikusan készült dokumentációval rendelkező programot fogadunk el.

A feladatokat a gyakorlatvezető osztja ki, többnyire azon feladatsor alapján, amelyet lejjebb olvashat.

Kiegészítő követelményeken egyrészt a nem-funkcionális követelményeket (például adatok beolvasásának módja, program újra indíthatósága stb.), másrészt a kódolási konvenciók betartását (például standard C++ elemek használata, előírt kódolási stílus), harmadrészt egyéb, a feladat megoldásához használt eszköz (például alprogramok használata, modulokra bontás, osztály használata) kötelező alkalmazását értjük. A kiegészítő követelmények halmozódnak, azaz egy újabb beadandó feladatnál érvényesek a korábbi feladatoknál előírt követelmények is.

A megoldáshoz elektronikusan készült dokumentációt kell készíteni, amelyet a gyakorlatvezetők kérhetnek kinyomtatva is. Nem kérünk teljes dokumentációt. Mindig az adott beadandó feladat-leírása határozza meg azt, hogy mi szerepeljen benne. (A mintadokumentáció ennél általában bővebb.)

A dokumentáció mindig tartalmazza tesztelési esetek listáját. Az egyes tesztesetekhez mindig meg kell adni egy konkrét adatsort és az erre várt eredményt is. Fájlból való olvasás esetén ezek a tesztesetek egy-egy fájlban legyenek meg.

A beadandó feladatokat 0-5 ponttal értékeljük. A beadási (bemutatási) határidőt a gyakorlatvezetők tűzik ki. Ha a gyakorlatvezető másképp nem rendeli, akkor a határidőre beadott, a követelményeknek megfelelő munka 5 pontot ér, késés esetén hetente egy ponttal kevesebbet. Minden munkát a szorgalmi időszakban kell bemutatni, és csak az kaphat gyakorlati jegyet, akinek minden munkája legalább 2 pontos lett.

Kiegészítő követelmények

Készítsen automatikus tesztet minden metódushoz, valamint a metódusok kombinálására is.

Dokumentáció

Tartalmazza a feladat szövegét, a megvalósított típus leírását, a típusműveletek megoldáshoz felhasznált programozási tételek neveit és a visszavezetésnél alkalmazott megfeleltetéseket, és a megoldó algoritmust, valamint a tesztelési esetek csoportokra bontott listáját.

Típus megvalósítása osztállyal

Az alábbi feladatok típusait egy-egy osztály segítségével valósítsa meg. Az összes megvalósítandó típus azonos típusú elemek sorozatát használja reprezentációként, amelyet tömbben (vector<>) kell eltárolni. Ahol a feladat szövege nem definiálja, az elemi típus az egész számok típusa. Egy osztály szolgáltatásainak (összes metódusának) bemutatásához olyan főprogramot kell készíteni, amelyik egy

menü segítségével teszi lehetővé a metódusok tetszőleges sorrendben történő kipróbálását. A főprogram példányosítson egy objektumot, amelyre a menüpontok közvetítésével lehessen meghívni az egyes metódusokat. Természetesen szükség lehet minden tevékenység után az objektum állapotának kiírására vagy egy az objektum állapotát kiíró külön menüpontra. Ha vannak olyan metódusok (esetleg barát függvények), amelyek több objektum közötti műveleteket valósítanak meg, a főprogram több objektum létrehozására és azok állapotának kiírására is adjon lehetőséget. Készítsen automatikusan futtatható teszteseteket is! Igyekezzen a reprezentációhoz olyan ötletet felhasználni, mely támogatja a műveletek hatékony megvalósítását!

Konkrét Feladat

Valósítsa meg az egész számokat tartalmazó zsák típust! Ábrázolja a zsák elemeit (az előfordulás számukkal együtt) egy sorozatban! Implementálja a szokásos műveleteket (elem betevése, kivétele, üres-e a halmaz, egy elem hányszor van a zsákban), valamint két zsák unióját (a közös elemek előfordulása összegződik), továbbá egy zsák kiírását!

Megvalósítás

Zsák Típus

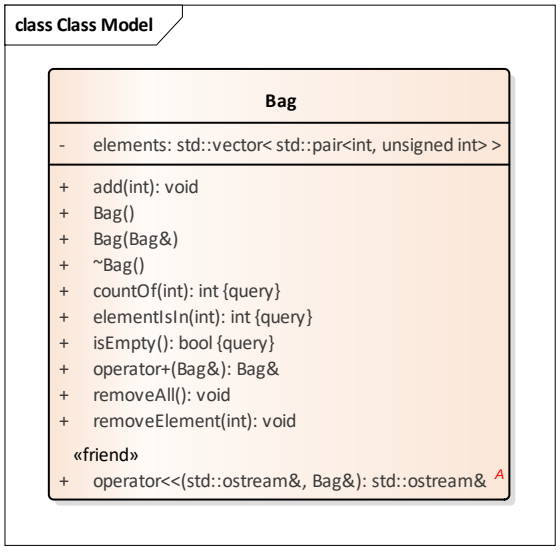
A Zsák típust egy osztály segítségével valósítjuk meg.

Reprezentáció

Az osztály belső reprezentációjához segítségül hívjuk a std::vector és az std::pair osztályokat. Mivel a zsákunkban „kulcs érték” párokat tárolunk, hiszen eltároljuk, hogy milyen egész számok vannak a zsákban, illetve minden számról, ami benne van, tároljuk azt is, hogy mennyi van belőle, ezért erre használjuk a Pair osztályt. A first-ben tároljuk, hogy milyen számot tárolunk, míg a second-ben tároljuk, hogy az adott elemből mennyit tartunk számon a zsákban.

A vectorban pedig ezeket a párokat fogjuk tárolni, így a zsák minden egyes elemét.

Osztály Diagram



Konstruktorok

Az osztályunknak összesen 2 konstruktora van.

Az első az alapértelmezett, ami nem vár semmit, mivel inicializálásra nincs szükség, mert a vektorunk képes saját magát inicializálni.

A másik pedig a másoló konstruktor, ami értékül kap egy másik Zsákot és az elements tartalmát lemásolja a vector és a pair osztályok másoló konstruktorjainak köszönhetően 1 sorban:

```
Bag::Bag(const Bag& b){
    this->elements = std::vector< std::pair<int, unsigned int> >(b.elements);
}
```

Destruktor

Mivel a belső reprezentációt Pair és Vector osztályok segítségével készítettük, így azok destruktorjai futnak majd le az osztály megszűnésekor, így teendőnk nincs.

Függvények

elementIsIn(int): int

Az első függvény megpróbálja megkeresni a függvény paraméterként kapott elemet a zsákban. Ha sikerül neki, akkor visszaadja annak indexét (, hogy a vektoron belül hol helyezkedik el), vagy, ha nem találja, akkor -1-el tér vissza. Tehát az elemünk akkor van benne a zsákban, ha >=0 számmal tér vissza.

```
const int Bag::elementIsIn(int e) const{
    for(int i = 0; i < elements.size(); i++){
        if(elements[i].first == e) return i;
    }
    return -1;
}
```

add(int): void

Az add() függvény segítségével tudunk számot betenni a zsákunkba, ezért kötelezően vár egy darab egész számot és azt teszi bele a zsákunkba.

Ahhoz, hogy egy egész számot a zsákba tegyünk a belső reprezentáció miatt meg kell vizsgálni, hogy benne van-e már a zsákban, mert, ha benne van (elementIsIn()>=0), akkor csak a darabszámot növeljük, egyébként pedig beletesszük úgy, hogy 1 db lesz a zsákunkban.

```
void Bag::add(const int e){
    int eIsIn = elementIsIn(e);
    if(eIsIn >= 0){
        elements[eIsIn].second += 1;
    }else{
        elements.push_back(std::pair<int,unsigned int>(e,1));
    }
}
```

countOf(int): int

Ez a függvény megvizsgálja, hogy a paraméterként kapott elemből hány darab van a zsákunkban. Az elementIsIn() függvény segítségével könnyen lekérhetjük annak helyét a vektorban (ha van), majd visszatérhetünk annak darabszámával. Ha nincs akkor 0-val térünk vissza.

```
const int Bag::countOf(int e) const{
    int eIsIn = elementIsIn(e);
    if(eIsIn>=0){
        return elements[eIsIn].second;
    }
    return 0;
}
```

isEmpty(): bool

Az isEmpty() függvényünk egy logikai értékkel tér vissza, hogy a zsákunk üres-e, vagy sem. true = üres, false = nem üres.

A vectornak hála ez könnyen kideríthető a következő módon:

```
const bool Bag::isEmpty()const{
    return elements.empty();
}
```

removeElement(int): void

A függvény paraméterként kapott egész számot megnézzük, hogy benne van-e a zsákunkban (az elementIsIn() függvény segítségével), ha nincs, akkor nem foglalkozunk vele, amennyiben viszont benne van, akkor további vizsgálatokra van szükség.

Ha több mint 1 darab van belőle a zsákban, akkor csak a darabszámot csökkentjük, viszont, ha csak 1 darab van benne, akkor ki is vesszük a zsákból, nem pedig 0 db-ként tároljuk, ezért is működik az előbb leírt elements.empty() függvényünk.

```
void Bag::removeElement(int e){
    int eIsIn = elementIsIn(e);
    if(eIsIn >= 0){
        if(elements[eIsIn].second > 1){
            elements[eIsIn].second -= 1;
        }else{
            elements.erase(elements.begin() + eIsIn);
        }
    }
}
```

removeAll(): void

A vektor osztály sokszínűségének megköszönhetjük, hogy a zsákunk ürítését szintén egy sorban elintézhetjük a clear paranccsal, mely törli a vector összes elemét, így a zsákunk tartalmát.

```
void Bag::removeAll(){
    elements.clear();
}
```

Operátor+

Az összeadás operátor 2 Zsák között működésbe lép és visszatér a zsákok uniójával tér vissza, ami egy új Zsák lesz.

Először is lefoglalunk számára a memóriában egy területet, mert nem szeretnénk, hogy a függvény lefutása után megszűnjön az unió zsákunk. Létrehozni pedig úgy hozzuk létre, hogy a már megírt másoló konstruktorunk segítségével a jobb oldali zsák tartalmát bemásoljuk az unió zsákjába.

Ezután pedig a bal oldali zsák összes elemét az add() függvény segítségével hozzáadjuk annyiszor, amennyi előfordul belőle.

Külső ciklus végigfut a zsák összes különböző elemén, a belső ciklus pedig annyiszor adja hozzá ahány darab van belőle.

```
const Bag& Bag::operator+(const Bag& b){
    Bag *tmp = new Bag(b);
    for(int i = 0; i < this->elements.size(); i++){
        for(int j = 0; j<this->elements[i].second; j++){
            tmp->add(this->elements[i].first);
        }
    }
    return *tmp;
}
```

Tesztelés

Automatikus Tesztelés

Az automatikus tesztelést egy BagTester osztály segítségével készítettük el, melynek összesen 6 tesztesete van és külön, külön meghívhatóak. Ezek mind előre megírt tesztesetek, csak a kód módosításával lehet őket megváltoztatni.

Használatukat a main függvényben a következőképp célszerű alkalmazni:

```
BagTester bt;
std::cout << "Test1 " << (bt.test1() ? "passed" : "failed") << std::endl;
std::cout << "Test2 " << (bt.test2() ? "passed" : "failed") << std::endl;
std::cout << "Test3 " << (bt.test3() ? "passed" : "failed") << std::endl;
std::cout << "Test4 " << (bt.test4() ? "passed" : "failed") << std::endl;
std::cout << "Test5 " << (bt.test5() ? "passed" : "failed") << std::endl;
std::cout << "Test6 " << (bt.test6() ? "passed" : "failed") << std::endl;
```

A függvények egy logikai értékkel térnek vissza, mely a teszt sikerességét jelzi (true = átment a teszteken, false = megbukott a teszteken).

Teszt 1:

Hozzáadunk egy számot a Zsákunkhoz, majd ellenőrizzük, hogy benne van-e.

Teszt 2:

A zsákhoz hozzáadunk 2 számot, majd törlünk belőle 1-et és megnézzük, hogy a törölt elem benne van-e a zsákban. Mivel a 2 szám amit hozzáadtunk különböző ezért a tesztünk akkor sikeres, ha nincs benne amit vizsgálunk.

Teszt 3:

Ebben a tesztünkben szintén 2 számot adunk hozzá a zsákunkhoz, majd törlünk minden elemet és ha a zsák ezután üres, akkor átment a teszten.

Teszt 4:

Ebben a tesztünkben 3 számot adunk hozzá, kétszer a 3-at, majd egyszer az 5-öt és ezután megnézzük, hogy a zsákban 2x szerepel-e a 3-as szám.

Teszt 5:

Utolsó előtti tesztünkben két zsákhoz is adunk hozzá számokat, melyek a hozzáadás után a következőket tartalmazzák:

Zsák 1: {3,2,2,2}
Zsák 2: {3,3}

Tehát az uniójuk valahogy így kell, hogy kinézzen: {2,2,2,3,3,3}

Létrehozzuk ezt az uniót és megvizsgáljuk, hogy valóban 3db 2-es és 3db 3-as van a zsákban.

Teszt 6:

Utolsó tesztünkben a Másoló konstruktor helyes működését teszteljük, mégpedig úgy, hogy hozzáadunk számokat egy zsákhoz, majd a másoló konstruktor segítségével létrehozunk egy új zsákot és ha ezek után tartalmazza mind azt, amit az először létrehozott zsákunk, akkor átment a teszten.

Manuális Tesztelés

A manuális teszteléshez szintént egy osztályt hívunk segítségül, így a main függvényben csak az osztályunk runManualTest() függvényét kell meghívnunk.

Mivel a későbbiek során is szükség lesz menürendszerre, ezért a Menüből a legtöbbet amit lehetett megcsináltam egy külön osztályként, mely arra képes, hogy hozzáadhatok menüpontokat és almenüket is a menünek, be is kéri, hogy mit szeretne választani a felhasználó, majd visszatér egy értékkel, ami pedig jelképezi, hogy a felhasználó melyik menüt választotta.

A manuális teszteléshez két zsákot használunk és a legtöbb menüpont mindkettőhöz hozzáférést ad.

Választási lehetőségek:

1. Új elemet hozzálehet adni valamelyik zsákhóz (felhasználó választja ki)
2. Elemet törölhetünk valamelyik zsákból
3. Minden elemet törölhetünk egy Zsákból
4. Megvizsgálhatjuk, hogy egy elem benne van-e egy zsákban
5. Megkérdezhetjük, hogy egy elem egy zsákban hányszor szerepel
6. Üres-e az adott zsák
7. Az unióját a két zsáknak kiírathatjuk (itt nincs almenü)
8. Kiírathatjuk a konzolra az egyik zsákot, amit a felhasználó kiválaszt.

Későbbiekben

Mivel a Menürendszert megjelenítő osztályunk elég kiforratlan ezért azt a következő két beadandó során még szeretném fejleszteni.