ADATSZERKEZETEK ÉS ALGORITMUSOK

Sor, Láncolt Lista

Iterátorok

- Speciális objektumok, általában egy gyűjtemény bejárására.
- Hasznos eszközök a felhasználói ciklikus léptetések elrejtésére, így sokkal tömörebb, érthetőbb kódot kapunk.
- Példa - STD tárolók használatánál:

```
vector<int> v(5);
vector<int>::iterator it = v.begin(); // az első elemre
                                      // mutató iterátor
```

Minden osztály definiálhat többféle iterátort is (pl.: reverse iterator).

Az iterátorok absztrakciója lehetővé teszi, hogy általános algoritmusokat írjunk, függetlenül a használt adattípustól.

Példa: átlagszámítás:

```
cout << accumulate(values.begin(), values.end(), 0.0) / values.size() << endl;
```

- Itt az `accumulate` függvény a `values` (amely valamilyen tároló típus) iterátorait használja.

`List` osztály - Iterátora

- A `List` osztály `Iterator` osztályának operátorai:

Operátor	**Funkció**
==	azonos-e a két iterátor
!=	különböző-e a két iterátor
++	a következő elemre lép
--	az előző elemre lép
*	dereferencia

- Ezekre az operátorokra van szükségünk az alapvető bejárás kivitelezéséhez
- Ezenkívül szükséges iterátor függvények:

```
Iterator begin() const; // a lista elejére állítja az iterátort
Iterator end() const;   // a lista végére állítja az iterátort
Iterator last() const;  // a lista utolsó elemére állítja az iterátort
```

Iterátorok – példa bejárásra

- Iterátorok használata lehetővé teszi az adatszerkezetünk egyszerű bejárását. Például:

```
for (typename List<T>::Iterator it = list.begin(); it != list.end(); ++it) {
  std::cout << *it << " ";
}
std::cout << std::endl;
```

- Az elemek tárolt értékének eléréséhez szükségünk van dereferencia (*) operátorra.

Iterátor osztály

- Egészítsd ki a `List` osztályt a belső `Iterator` osztályával, illetve az új `find` metódussal!

```
Iterator find(const T &e) const;
```

- Készíts saját kiíró operátort (Az iterátor segítségével, lásd `list.hpp` fájl vége)!
  - `cout << list;`

STL

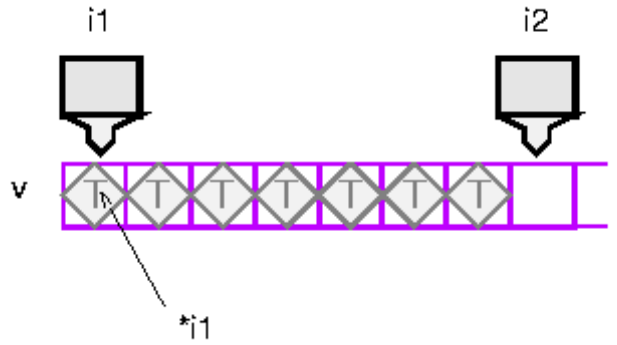
- Tárolók
  - vector, dequeue, stack, map, és sok más
- Iterátorok
  - A tárolók belső típusai amik segítségével bejárható az adatszerkezet
- Algoritmusok
  - Gyakran használt algoritmusok, melyek tárolótól függetlenül végrehajthatók
    - (template függvények)
  - Átlagszámolás példa:

```
cout << accumulate(values.begin(), values.end(), 0.0) / values.size() << endl;
```

STL iterátorok

- Az STL algoritmusai általánosak, bármilyen típuson és tárolón működnek. Ezt az *iterátorok* teszik lehetővé
  - Egy iterátorral meghatározható egy pozíció a tárolóban
  - Növelhető (léptethető), dereferálható (a pozícióban tárolt érték eléréséhez) és két iterátor összehasonlítható
  - Van egy speciális “vége” iterátor: az utolsó elem utánra mutat.
  - A tárolóktól lekérhető az első és a “vége” iterátor.

```
vector<int> v;
vector<int>::iterator i1 = v.begin();
vector<int>::iterator i2 = v.end();
vector<int>::iterator i3 = v.begin()+4;
```



STL iterátorok

- Az olyan algoritmusok mint az `accumulate` vagy a `sort` két iterátort fogadnak paraméterként: az eleje és a vége.
  - A végrehajtás során az eleje iterátort növelik és dereferálják, egészen addig amíg a vége iterátorral egyenlő nem lesz.

- Lehetséges `accumulate` implementáció:

```
template <class InputIterator, class T>
T accumulate (InputIterator start, InputIterator stop, T initial) {
  while(start != stop) {
    initial += *start;
    ++start;
  }
  return initial;
}
```

- Többféle iterátor kategória van, aszerint hogy a tároló milyen műveleteket támogat (pl véletlen elérés, hátralépés, stb.).

C++ lambdák

- Hasonló koncepciók: lambda, closure, anonymus function
  - Függvény objektum: olyan objektum, amelynek meg van írva a () operátora.
  - C++ lambda: olyan függvény objektum, amit “inline” definiálunk – így nincs szükség külön kódrészletben a típus deklarációjára.
  - Szintaxis:

```
[ captures ] (parameters) -> returnTypesDeclaration { lambdaStatements; }
```

    - `captures`: mely, a bennfoglaló kontextusban lévő változók lesznek elérhetők a lambdán belül – érték vagy referencia szerint
    - `parameters`: argumentumok listája
    - `returnTypesDeclaration`: visszatérési érték – általában kihagyható
    - `{...}` a függvény törzse

Példa 1:

```
auto lambda = [](){ cout << "Code within a lambda expression." << endl; };
lambda();
```

Példa 2:

```
auto sum = [](int x, int y){ return x + y; };
cout << sum(5, 2) << endl;
cout << sum(10, 5) << endl;
```

for_each

- Egy tárolón való végigiterálás újabb lehetséges módja

```
struct Sum {
    Sum(): sum{0} { }
    void operator()(int n) { sum += n; }
    int sum;
};
int main() {
    std::vector<int> nums{3, 4, 2, 8, 15, 267};
    auto print = [](const int& n) { std::cout << " " << n; };
    std::cout << "before:";
    std::for_each(nums.begin(), nums.end(), print);
    std::cout << '\n';
    std::for_each(nums.begin(), nums.end(), [](int &n){ n++; });
    // calls Sum::operator() for each number
    Sum s = std::for_each(nums.begin(), nums.end(), Sum());
    std::cout << "after: ";
    std::for_each(nums.begin(), nums.end(), print);
    std::cout << '\n';
    std::cout << "sum: " << s.sum << '\n';
}
```

Gyakorló feladat I.

- Terheljük túl a lista += operátorát. Egy meglévő kétirányú láncolt lista végére fűzze hozzá a kapott kétirányú láncolt lista elemeinek másolatát.
- Terheljük túl a lista + operátorát. Egy meglévő kétirányú láncolt lista másolatának végére fűzze hozzá a kapott kétirányú láncolt lista elemeinek másolatát és adja vissza visszatérési értékként.
- Írjunk egy cut() függvényt, ami kettévágja a jelenlegi listát. A cur utáni részlistából csináljunk egy új, önálló listát és adjuk vissza visszatérési értékként.

Gyakorlófeladat G03F03

- Adott egy fájl amiben titkosírással elrejtettünk egy üzenetet. Verem és sor használatával, fejtsd vissza mit titkosítottunk.
- A fájlban #, & és @ szimbólumok valamint a következő karakterek szerepelhetnek: a-z, A-Z, 0-9
- Minden karakter két szimbólum közé esik.
- A megfejtés azon karakterek (a fájlban szereplésüknek megfelelő sorrendben) összeolvasása amelyeket két azonos szimbólum határol.
- Példa a fájlból:
  - #a&#l@#Q@&A##v@@7#@D&#y&@k&@8##W@&V@
  - &i@#H#@V&@q&@s#@D#@q&@F##s&&v##e@#L&#a&
  - &7##p&&J@#i&&n@@R##M@@f#&j@&V##Y&&T@&4##l
- Az eredményt írd ki a konzolra és egy fájlba.

Gyakorlófeladat G03F04

- Írj a hallgatók beiratkozását szimuláló programot!
- A beiratkozás során a hallgatók érkezési sorrendben, az éppen aktuálisan szabadon lévő TO-s előadóknál iratkoznak be: Bokhara-Rigli Etánál, Szini-Társ Hant Évánál, és Rumba-Rigli Leánál.
- A program feladata, hogy feljegyezze a beiratkozás sorrendjét a három TO-s előadónál.
- A hallgatók listája érkezési sorrendben egy in.txt nevű fájlban van elmentve, innen kerülnek be a sorba. Mindenkinek véletlenszerűen 1-5 percig tart a beiratkozás.
- Az out.txt-ben mentsd el, hogy hogyan zajlott a beiratkozás. Minden hallgató külön sorba kerüljön, és szerepeljen az is, hogy kinél, és hány percet töltött.
- Pl.: Kis Eufrozina Rumba-Rigli Leanal 5 perc alatt iratkozott be.

Gyakorlófeladat G03F05

- Készítsd el a deque implementációját a statikus sor módosításával. A deque kétirányú sor, tehát mindkét végére tehetünk be és vehetünk ki onnan elemet, de a középső elemeket nem érjük el.
- A deque működésének bemutatásához készíts egy piros-zöld papucs kártyajátékot szimuláló programot. (Ez a piros papucs nevű kártyajáték kicsit módosított változata.)
- A játékot két játékos játssza egy pakli (32 lapos) magyar kártyával. A kártyalapok értéke nem, csak a színe (piros, zöld, tök, makk) számít.
- A lapokat először szétosztják maguk között két megkevert, egyenlő méretű pakliba (figyelj rá, hogy minden színből 8 db legyen). Ezután felváltva pakolnak kártyát a saját paklijuk **tetejéről** az asztalon levő kupac **tetejére**.
- (folyt. köv.)

Gyakorlófeladat G03F05 (folyt.)

- Ha valaki *pirosat* rakott, akkor a másik felveszi az asztalon levő teljes paklit, és sorban elhelyezi a saját paklija **aljára**, **felülről lefelé**. Ha valaki *zöldet* rakott, akkor szintén a másik veszi fel a teljes paklit, de fordított sorrendben (**alulról felfelé**) helyezi el a saját paklija **aljára**.
- A játéknak vége, ha elfogyott valamelyik játékos paklija, és ő következik soron; ekkor ő nyert (azaz lehet „visszahívni” 1 körig). Ha 200 lépés után sem fogyott el senkinek a kártyája, akkor a játékosok megunják, és kiegyeznek egy döntetlenben.
- A megvalósításban a játékosok kezében levő paklikat egy-egy sorral, az asztalon levő paklit pedig egy kétirányú sorral valósítsd meg! A kártyák számértékét nem szükséges figyelembe venned.

Gyakorló feladat G03F06

- Pályaudvar szimulátor
- A pályaudvar fix darabszámú vágányból áll
- Egy vágányon egyszerre több vonat is tartózkodhat
- Egy vágányról az a vonat indulhat el először (értelemszerűen), aki elsőnek érkezett
- A pályaudvar vágányaira vonatok érkeznek, amelyek különböző darabszámú vagonnal rendelkeznek
- Egy vagon tartalma szöveges. pl.: 'biciklis kocsi', 'marhavagon', 'első osztály', stb. Nem kell külön osztály a vagonoknak!
- A vonathoz bárhova hozzá lehet illeszteni egy új vagont. Kiválasztjuk az aktuális vagont, és elé, vagy utána új vagont illesztünk be

Gyakorló feladat G03F06 (folytatás)

- A program véletlenszerűen érkeztet néhány vonatot a pályaudvarra (Egy vágányra több vonat is érkezzen!)
- Néhány vágányon egy-egy vonathoz tegyen a program egy új vagont! (pl. az első vágányon levő második vonat végéhez hozzáteszi a 'gyerekkocsi' vagont.)
- Néhány véletlenszerű vágányról elindít egy-egy vonatot!

Gyakorló feladat G03F07

- Egyszerű adatbázis
  - Készíts egy adatbázis programot, ami egy eltárolt fájlt képes beolvasni, a benne levő elemeket módosítani, majd visszaírni a fájlba.
  - Az adatbázisban tárolt elemek egyszerű string-ek.
  - A fájlban az elemeket soronként tárold.
  - A program induláskor beolvassa a fájl tartalmát egy listába.
  - Ezután egy konzolos menü interfészt ad, amelyen keresztül a felhasználó kiírathatja az összes elemet, eltávolíthat, hozzáadhat.
  - A program kilépéskor írja be az elemeket a fájlba, így következő induláskor megint láthatóak a tárolt elemek.

Gyakorló feladat G03F08

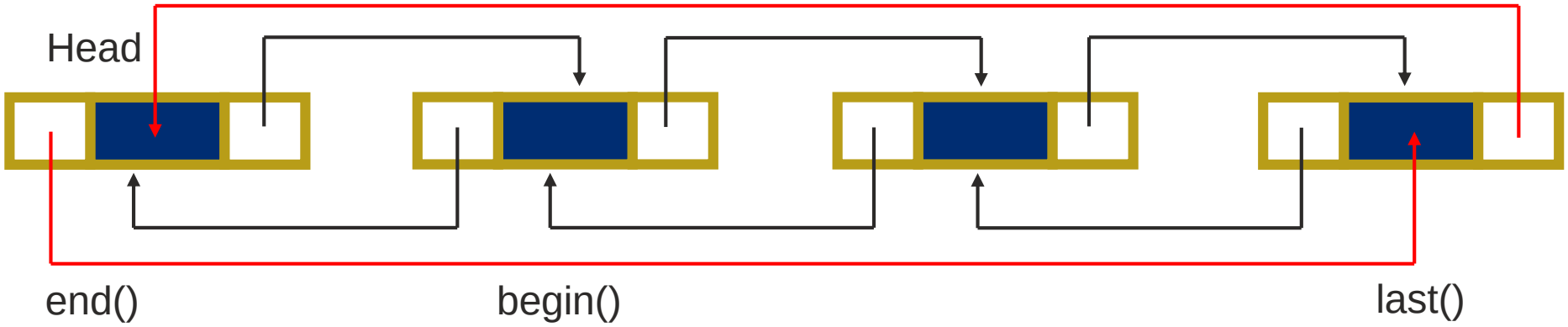
- Lista statikus implementációval
  - A lista elemeit nem dinamikus láncolással, hanem statikusan, egy tömbben tárold.
  - A tömb elemei érték-index párok.
  - A lista az összes lista műveletet implementálja.
  - Iterátorokkal is rendelkezzen.
  - Copy constructor és assignment operator is legyen rajta definiálva.
  - Figyelj arra, hogy a statikus ábrázolás miatt a lista nem csak alul-, hanem túl is csordulhat!

Gyakorló feladat G03F9

- Írj egy befűző függvényt, amely paraméterül kap egy másik listát,és az elemeit befűzi az aktuális elem utánra (és írd meg azt is ami az aktuális elem elé szúrja be)!
  - A függvény legyen a lista tagfüggvénye.
  - Profiknak: A függvény ne legyen része az alap listának! Ezen kívül a befűzés konstans időben fusson le, és a befűzendő listát ürítse ki!
- Ezek után használd a listát mint stringet, tehát tárolj benne karaktereket (`char`)!
- Kérj be a konzolból egy szöveget, és töltsd fel vele a listát!
- Ezután kérj be egy másik szöveget, és egy karaktert, majd a szöveget fűzd be az első szövegbe úgy, hogy az a bekért karakter első előfordulása után kezdődjön!

Gyakorló feladat G03F10

- Valósítsunk meg kétszeresen, körkörösen láncolt listát fejelemmel!
  - A körkörös láncolás azt jelenti, hogy a `begin()` iterátor a fejelem rákövetkezőjére mutat, az `end()` a fejelemre, a `last()` pedig a fejelem megelőzőjére



Ez azt eredményezi, hogy a lista végén nincs nullpointer, hanem a láncot „visszaakasztjuk” a lista elejére

Az `insert` művelet így lényegesen leegyszerűsödik, hiszen nem kell vizsgálni a külön eseteket, minden elemnek van előző és következő eleme

3. Kis házifeladat

- Modellezd egy betegellátó egység működését a következő módon!
- A betegek sorban állnak a kezelésért. Aki a sor elején áll azt kezelik először, utána a következőt és így tovább.
- Azonban a betegellátó személyzet dönthet úgy, hogy először a gyerekeket kezeli. Ekkor a sor elejére kerülnek a gyerekek a sorrendjüknek megfelelően.
- Hasonlóan dönthet a személyzet arról, hogy először a nőket vagy a férfiakat kezelik. Továbbra is a rendezés után a nők (vagy a férfiak) egymáshoz viszonyított sorrendje megegyezik.

3. Kis házifeladat

- Példa: A sorba beállnak a betegek. Először Éva (nő), majd András (férfi) ... végül Virág (nő). Így a sor hátulról kezdve:
  Virág (nő), Anna (nő), Béla (gyerek), Jenő (férfi), Evelin (gyerek), András (férfi), Éva (nő)

  Elhangzik az utasítás, hogy a gyerekeket kezeljék először. Ekkor a sor hátulról kezdve:
  Virág (nő), Anna (nő), Jenő (férfi), András (férfi), Éva (nő), Béla (gyerek), Evelin (gyerek)
  (Figyelj rá, hogy a gyerekek egymáshoz viszonyított sorrendje nem változik!)

  Beáll egy új beteg a sorba: Ilona (gyerek). Ekkor a sor hátulról kezdve:
  Ilona (gyerek), Virág (nő), Anna (nő), Jenő (férfi), András (férfi), Éva (nő), Béla (gyerek), Evelin
  (gyerek)

  Elhangzik ezután az utasítás, hogy a nőket kezeljék először. Ekkor a sor hátulról kezdve:
  Ilona (gyerek), Jenő (férfi), András (férfi), Béla (gyerek), Evelin (gyerek), Virág (nő), Anna (nő),
  Éva (nő)

  Így először Évát (nő), majd Annát (nő) ... végül Ilonát (gyerek) kezelik.

3. Kis házifeladat

- Specifikáció: A sorba egyesével érkeznek a betegek. A beteg neve és csoportja ismert (gyerek, nő, férfi). Valósítsd meg ezt az adatszerkezetet a következő módon:
  - Adatszerkezet neve: hospitalQueue, amely egy dinamikus méretű sort valósít meg
  - Adatmezői:
  - name - beteg neve (string)
  - group - beteg csoportja (gyerek, nő vagy férfi) (string)
  - Függvényei:
  - in(name, group) - sorba egy személy beáll, megadva a személy nevét és csoportját
  - out() - sor eleji személy kezelése, visszatérési értéke a személy neve, ha üres a sor, akkor UnderFlowException-t dob
  - order(group) - csoportosítás módja (nő, gyerek vagy férfi)
  - isEmpty() - a sor üres-e
  - first() - a sor elején álló személy neve