Dinamikus láncolt lista

4. GYAKORLAT

Szekvenciális adatszerkezet

A szekvenciális adatszerkezet olyan <A, R> rendezett pár, amelynél az R reláció tranzitív lezártja teljes rendezési reláció.

Szekvenciális adatszerkezetben az egyes adatelemek egymás után helyezkednek el, van egy **logikai sorrendjük**.

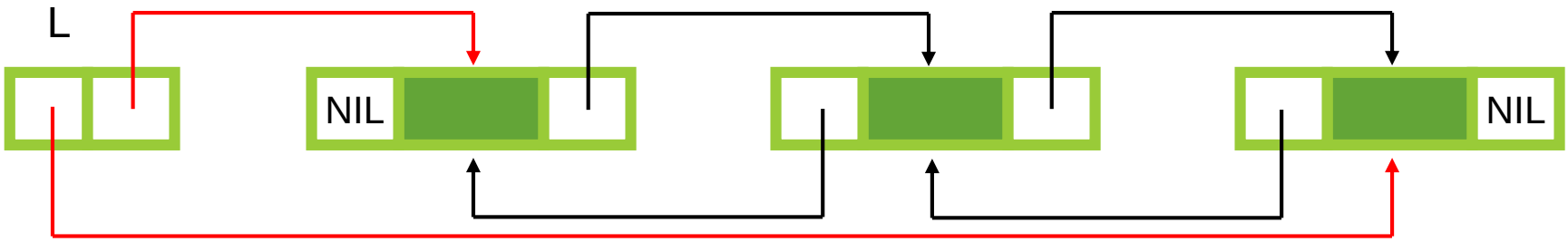
Az adatok között egy-egy jellegű a kapcsolat: minden adatelem alapvetően csak egy helyről érhető el és az adott elemtől csak egy másik látható.

Két kitüntetett elem: az **első** és az **utolsó**.

Láncolt listák

Egyirányú láncolt lista
pl.: A láncolt dinamikus sor nagyon hasonló egy ilyen listához
(Csak a műveletek mások.)

Kétirányú láncolt lista
(Ezen a gyakorlaton ezzel fogunk foglalkozni)



Node osztály

```
class Node {
public:
   int data;               // Az adat, amelyet tárol
   Node* prev;             // Az előző elemre mutató pointer
   Node* next;             // A következő elemre mutató pointer
   Node(const int& data); // konstruktor
   Node(const int& data, Node* prev, Node* next); // konstruktor
};
```

Mivel láncolt ábrázolásról van szó, kell egy `Node` osztály, ami tartalmazza az értéket, és hivatkozást a szomszédos elemekre.

A `head` megelőzője és a `tail` rákövetkezője `NULL`.

Kétirányú láncolt lista osztály

```
class List {
   private:                  // Ez a default láthatóság, de szebb, ha kiírjuk
      class Node {
         …
      };

   public:
         …
};
```

A `Node` osztály a listának egy belső osztálya (enkapszuláció), és mivel `private` a listán belül, a listán kívüli programrész nem is látja, nem is tud a létezéséről. Csak a belső szerkezet megvalósítására használjuk.

A `Node`-nak minden adattagja `public`, ami azt jelenti, hogy a lista bármikor láthatja a `Node` belsejét.

Láncolt lista elemei

```
class List {
   protected:
      Node *head; // A lista eleje
      Node *tail; // A lista vége
      Node *act;  // Az aktuális elem
};
```

Három kitüntetett elem van. A lista eleje, a vége és egy 'aktuális' elem.

Ezzel az aktuális elemmel járhatjuk be az egész listát.

Láncolt lista metódusai

Nyissuk meg a **DinLinkedList_int** project **list.hpp** fájlját

Nézzük át közösen a láncolt lista metódusainak specifikációját

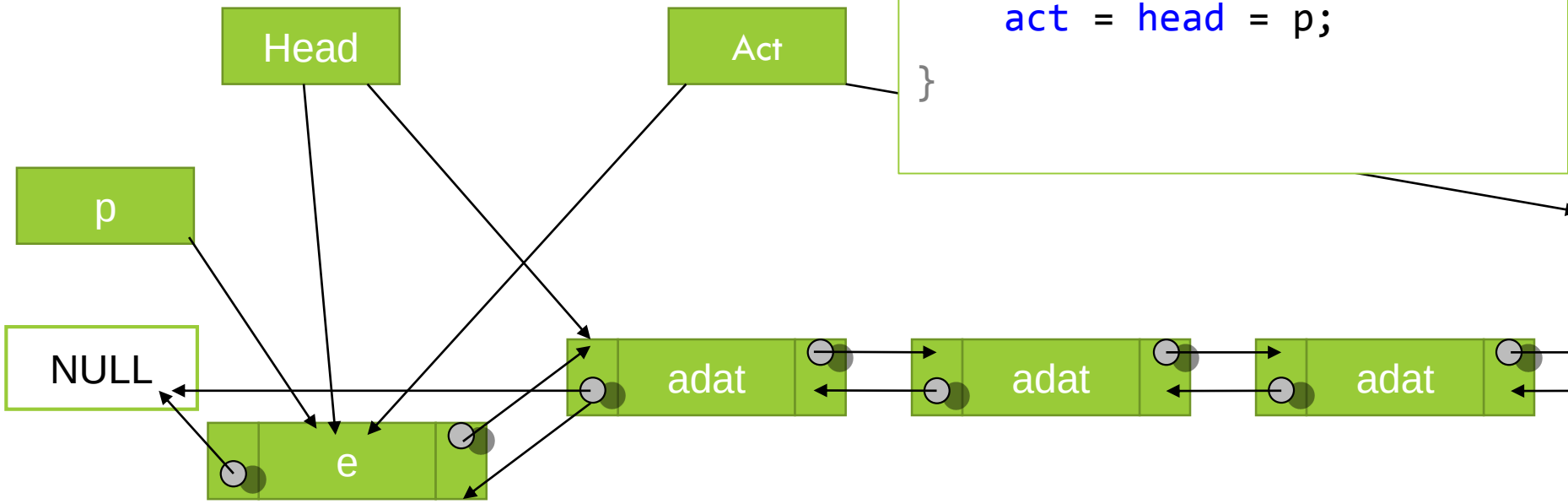
Írjuk meg közösen a következő három dián bemutatott metódusokat

Láncolt lista - insertF

Példa 1: **insertFirst(int e)**

```
Node* p = new Node(e);


if (isEmpty()) {
    act = head = tail = p;
} else {
    p->next = head;
    head->prev = p;
    act = head = p;
}
```

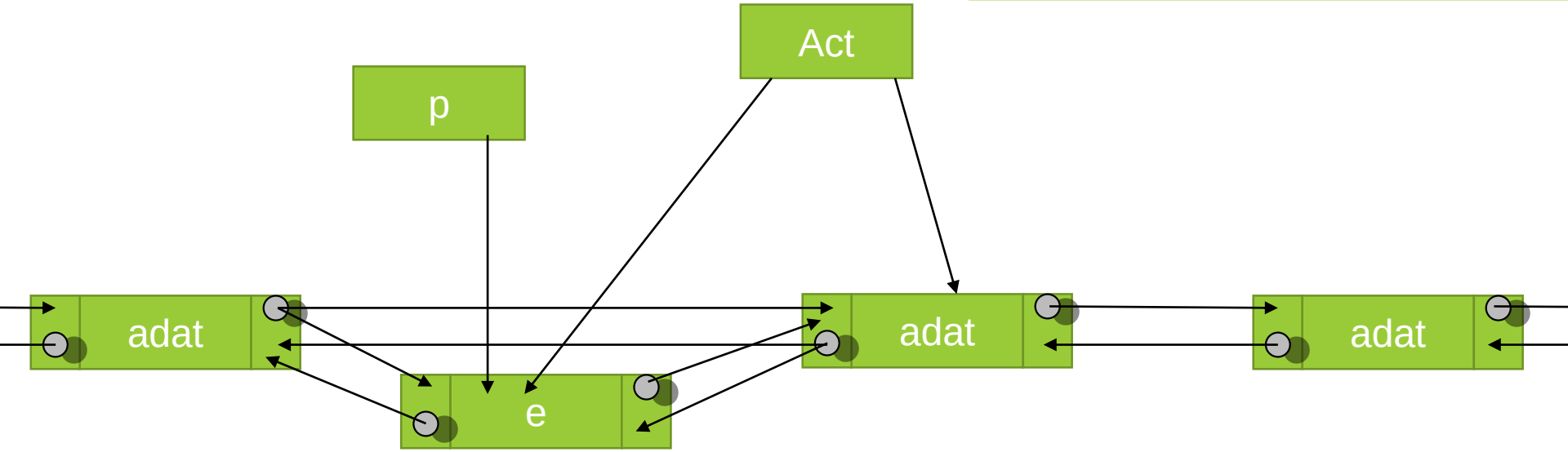


Láncolt lista - insertBe

Példa 2: **`insertBefore`**`(`**`int`**` e)`

```
Node* p = new Node(e,
act->prev, act);

act->prev->next = p;
act->prev = p;
act = p;
```

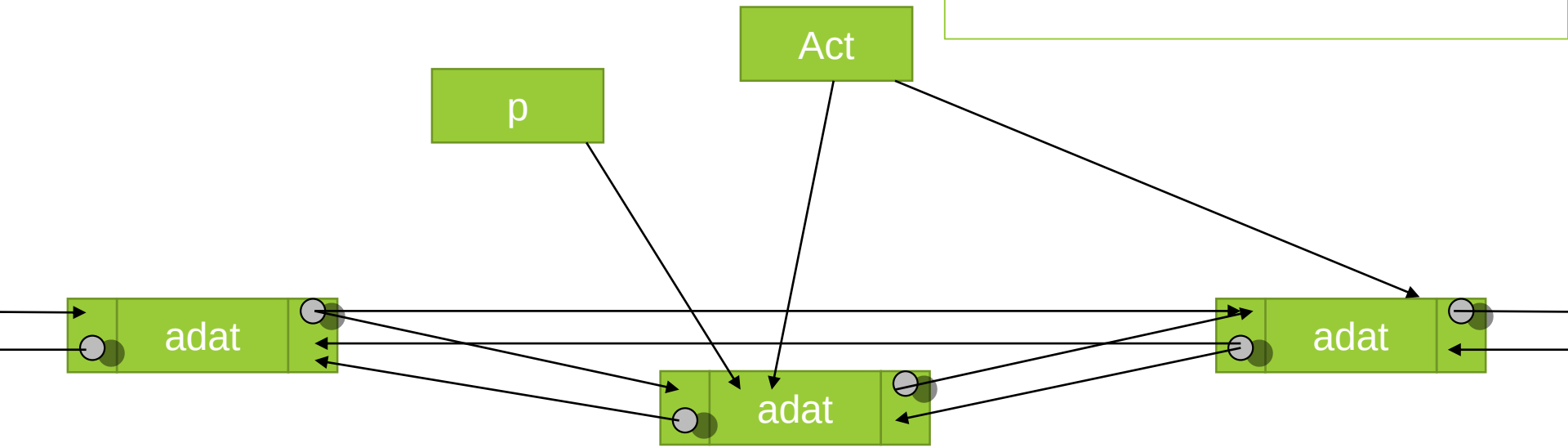


Láncolt lista - remove

Példa 3: **`removeAct()`**

```
Node* p = act;

act = act->next;
p->next->prev = p->prev;
p->prev->next = p->next;
delete p;
```



Láncolt lista

A megírt metódusok alapján írjátok meg a hiányzó metódusokat!

- `insertLast`
- `insertAfter`
- `removeFirst`
- `removeLast`

String-eket tartalmazó lista

Nyissátok meg a **DinLinkedList_string** projectet

Másoljátok át a **list.hpp** fájlt ebbe a mappába és adjátok hozzá a projecthez

Módosítsátok, hogy `string`-eket lehessen betölteni a listába

Teszteljétek a mellékelt main.cpp-vel

String vs. int

Szinte ugyanaz a kód

Biztos van valami jó megoldás...

Sablon (Template)

Egy függvény vagy osztály definíciója során megtehetjük, hogy a paraméterek, változók vagy épp a visszatérési érték típusát nem egy előre ismert típusként adjuk meg, hanem egy típust jelző változóval helyettesítjük (típusparaméter).

Tehát az adott kódot csak egyszer kell megírnunk és az több típus kezelésére is alkalmas lesz sablonok használatával.

Template - példa

```
void swap(int &a, int &b) {
    int c = a;
    a = b;
    b = c;

}
…
int a(3), b(4);
swap(a, b);
```

```
template <typename T>
void swap(T &a, T &b) {
    T c = a;
    a = b;
    b = c;
}
…
int a(3), b(4);
swap<int>(a, b);
swap(a, b); //spec. eset
```

Template

Template-tel paraméterezett függvény hívásakor, ha a fordító *egyértelműen* el tudja dönteni a típusparamétert, nem szükséges explicit módon megadni. (A példában a második függvényhívás: `swap(a,b);`).

```
template <typename T>
void f() {
    …
}
```

```
template <typename T>
void g(T x) {
    …
}
```

Template

Template-tel paraméterezett kód esetén a fordító a használat alkalmával készíti el az adott típushoz tartozó tárgykódot.

Azaz a `swap<int>(a, b);` és a `swap<long>(a, b);` hívások során két külön verzió készül a `swap` függvény kódjából.

Emiatt minden template-tel paraméterezett kódnak a header file-ban kell lennie.

Template - osztály

A template-es osztályok szintaxisa a következő:

```
template <typename T>
class A {
    T data;
    T f();
};
```

```
template <typename T>
T A<T>::f() {

}
```

Amint látható, a megvalósításnál is jeleznünk kell a template paramétert

A deklarációnak és a megvalósításnak itt is egy fájlban kell lennie

A két relációs jel között a **`typename`** és a **`class`** kulcsszó általában ugyanazt jelenti

Template

Jó tudni: C++-ban nem tudunk megszorításokat tenni a típusparaméterre. Ennek következménye, hogy, ha a típusra nem létezik egy adott művelet/változó/stb., akkor a problémához csak részben kapcsolódó, nehezen értelmezhető hibaüzenetet kapunk.

Általános kétirányú láncolt lista

Nyissátok meg a **DinLinkedList_template** projectet

Írjuk át a meglévő **list.hpp** fájlokat sablonos verzióra

Általános kétirányú láncolt lista

Továbbra is a **DinLinkedList_template** projectet nézzük

Írjuk meg az új `clear()` segéd eljárást.

Most már tesztelhetjük a **main.cpp**-vel

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

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 (List<int>::Iterator it = list.begin(); it != list.end(); ++it) {
        cout << *it << ' ';
}
cout << endl;
```

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

Iterátor osztály

Nyissuk meg a **DinLinkedList_iterator** project **list.hpp** fájlját

Nézzük át a kódot és közösen készítsük el az iterátor osztályt és metódusait.

Készítsük el az új lista metódust is, mely a keresett elemre mutató iterátort adja vissza

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

Készítsünk egy saját kiíró operátort (Az iterátor segítségével, lásd **list.hpp** fájl vége)!

- `cout << list;`

Gyakorló feladat G04F01

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 G04F01 (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 G04F02

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 G04F03

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 G04F04

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`)!

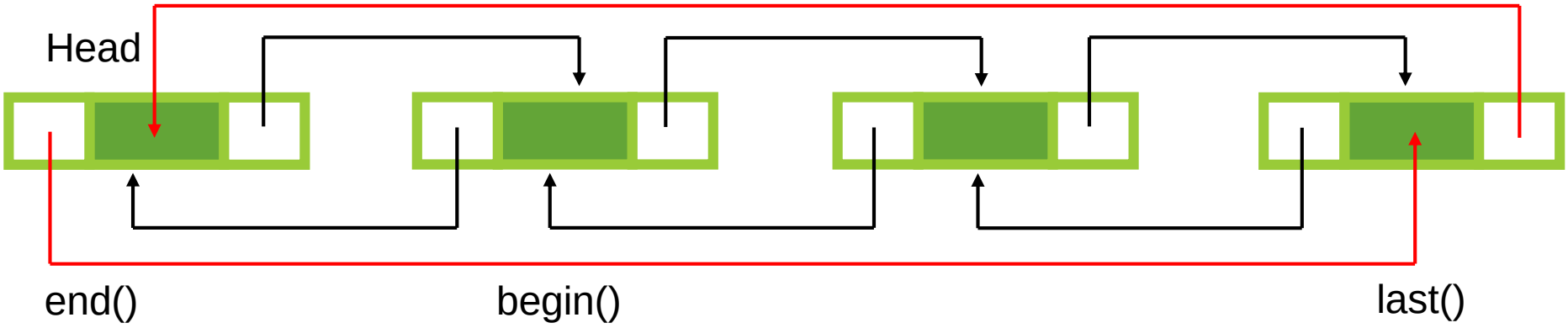
Kérj be a konzolból egy szöveget, és töltsd fel vele a listát!

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 G04F05

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

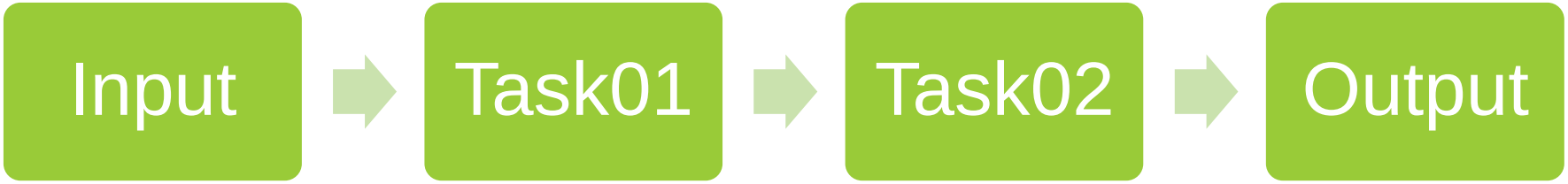
Az `insert` művelet így lényegesen leegyszerűsödik, hiszen nem kell vizsgálni a külön eseteket, minden elemnek van előző és következő eleme

Házi feladat G04F06

A feladat egy gyártósor felépítése. A sor egyes állomásait node-ok reprezentálják, amik egyéni (elemi) feladatokat tudnak elvégezni. De ahhoz, hogy egy termék elkészülhessen ezen alegységek együttműködésére van szükség.

Legyenek előre definiált feladatokat elvégző node-ok. Egy szöveges fájlból lehessen megadni az egyes alfeladatokat. A szöveges fájl alapján építsük fel a feladatot elvégezni képes gyártósort, majd végezzük el a feladatot.

Az első node fogadja a bemenő paramétereket (input node), az utolsó node pedig megadja a végleges terméket (Kiírja a paramétereit egy szöveges fájlba.)



Házi feladat folyt. G04F06

Feladat:

Autó összeszerelő gyártósor:

- Szükséges node-ok: karosszéria gyártó, elektronika beszerelő, festő, …

A gyártósor egy tényleges autó objektumon dolgozzon. Az egyik node adjon hozzá egy motor objektumot az autóhoz (lehessen paraméterezni: sebesség, gyári szám), a másik rendeljen hozzá színt és rendszámot. Majd végül az output node jelenítse meg az autó paramétereit.

Bemeneti txt péda: // input node feldolgozza az adatokat

Audi // karosszéria node létrehozza az autót (súlya is van a karosszériának)

140,100 // beszereli a létrehozott motor objektumot (sebesség, gyári szám)

Fekete,ABC-001 // színt ad az autónak és rendszámot

Kulcsszavak: polimorfizmus, dinamikus kötés, láncolt lista

Házi feladat folyt. G04F06

Házi feladat hint:

Az órán megírtunk egy template-es láncolt listát. Ezt fel tudjuk használni mégpedig úgy, hogy Egy ős szerelő osztály típusú listát hozunk létre.

LinkedList<Szerelo> list; // a data adattag Szerelő típusú lesz

Ennek a szerelő ősosztálynak van egy pl.: doSomething() metódusa, amit a leszármazottaknak felül kell definiálni. Tehát az Input node beolvas egy fájlt. A motor szerelő létrehozza a motort és beszereli. …

Mikor felépült a lista, akkor meghívunk pl.: egy start() metódust, vagyis bejárjuk a listát, miközben minden egyes nodnak lefut a saját doSomething() implementációja és elkészül a termék.