

# HÁZI FELADAT

## Programozás Alapjai 2

### Feladatválasztás/Feladatspecifikáció

Zagyvai Péter

HDUEO8

2024.04.21

## TARTALOM

1.)	<a href="#">Feladat</a> .....	2
2.)	<a href="#">Feladatspecifikáció</a> .....	2
3.)	<a href="#">Terv</a> .....	3
3.1)	<a href="#">Objektum modellek</a> .....	3
3.1.1)	<a href="#">Lista Elem</a> .....	3
3.1.2)	<a href="#">Lista</a> .....	3
3.1.3)	<a href="#">Iterátor</a> .....	5
3.1.4)	<a href="#">Teszt Osztály</a> .....	6
3.1.5)	<a href="#">Összehasonlító Functorok</a> .....	6
3.1.6)	<a href="#">Teljes UML-diagram</a> .....	7
3.2)	<a href="#">Algoritmusok</a> .....	8
4.)	<a href="#">Megvalósítás</a> .....	13
4.1)	<a href="#">Tesztprogram bemutatása</a> .....	13
4.2)	<a href="#">Futtatott tesztesetek</a> .....	13
4.2.1)	<a href="#">Egész számokon futtatott tesztesetek</a> .....	13
4.2.2)	<a href="#">C Stringeken futtatott tesztesetek</a> .....	13
4.2.3)	<a href="#">Saját Osztályon futtatott tesztesetek</a> .....	14
4.3)	<a href="#">Memóriakezelés</a> .....	14

## 1. Feladat:

### Generikus Lista

Készítsen generikus duplán láncolt rendezett listát! A kulcsok közötti rendezettséget a szokásos relációs operátorokkal vizsgálja, amit szükség esetén specializál! Használjon strázsát a lista elején!

Valósítsa meg az összes értelmes műveletet operátor átdefiniálással (overload), de nem kell ragaszkodni az összes operátor átdefiniálásához! Legyen az osztálynak iterátora is! Legyen képes az objektum perzisztens viselkedésre!

Specifikáljon egy egyszerű tesztfeladatot, amiben fel tudja használni az elkészített adatszerkezetet! A tesztprogramot külön modulként fordított programmal oldja meg! A megoldáshoz ne használjon STL tárolót!

## 2. Specifikáció:

A feladat egy generikus lista elkészítése. A lista képes lista elemek tárolására, felvételére és az elemek lekérdezésére, elemek rendezésére.

### 2.1. Elem Felvétele

A listába új elem felvételére három mód áll rendelkezésre:

- 1.) Lista elejére vesszük fel az új elemet.
- 2.) Lista végére vesszük fel az új elemet.
- 3.) Lista egy adott helyére (index) vesszük fel az új elemet, azt csak akkor tudjuk megtenni, ha valid indexet adunk meg (0-nál nagyobb, lista méreténél nem nagyobb).

### 2.2. Elemek Elérése

Lista elemeit akár a [] operátorral vagy akár függvénnyel is elérhetjük. Mindkét esetben index megadásával tudjuk lekérni a kívánt elemet, illetve szintén mindkét esetben működik konstans tagokra is. (Csak valid indexelés esetén működik)

### 2.3. Iterátor

Lista elemei továbbá elérhető iterátorral is. Az iterátor képes:

- 1.) Előre/Hátra haladni, post- és pre-növelés/csökkentés operátorokkal.
- 2.) Lista elem perzisztens viselkedésére a „->” és „\*” operátorokkal.

Továbbá az iterátor ad lehetőséget a lista bejárására.

### 2.4. Lista rendezése

A lista alpból nem tárolja rendezetten az elemeit (lista elemek felvételének módja miatt ez nem lehetséges), de ad lehetőséget a listaelemek rendezésére és a duplikált lista elemek törlésére is.

### 2.5. Lista elemeinek írása/olvasása

Mivel ez egy generikus lista, ezért az egyes fajta elemeket írni/olvasni nem lehet ugyanazzal a módszerrel. Írásról és olvasásról az egyes class-oknak kell gondoskodniuk. A tesztesetek során lesz példa egy egyszerű integer és egy összetett class (1 bool és 1 int) írására és olvasására.

### 3. Terv

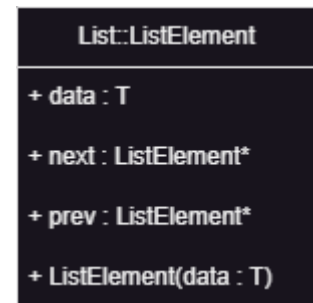
A feladat 4 class (Lista, Lista Elem, Iterátor, Teszt Class) és egy tesztprogram megtervezését igényli.

#### 3.1. Objektum Modellek

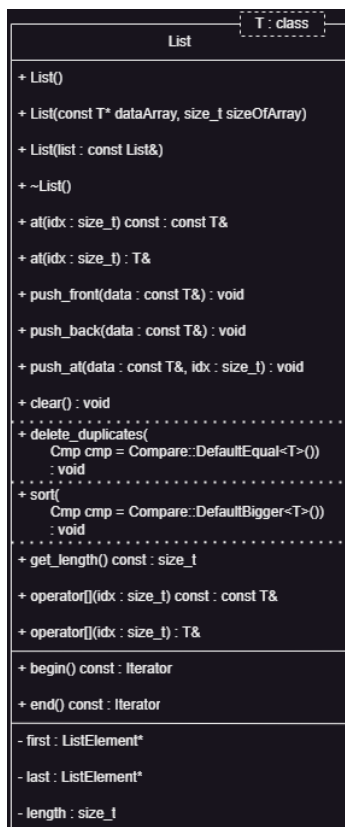
A lista alapjaiban generikus lista elemekből épül fel. (Az egyes algoritmusok a 3.2.-es részben találhatóak meg kifejtve.)

##### 3.1.1. Lista Elem

A Lista Elem egy privát osztály a Lista namespace-en belül. Ebből kifolyólag a lista elemen belül nincs szükség az adatok elrejtésére, ugyanis azok csak a lista osztály számára lesz elérhető, ezért minden adat publikus. A lista elem tárolja az előző és következő lista elemet (azoknak a pointerait) és tárol egy generikus adatot. Ha valamelyik pointer NULL értékre mutat, akkor az jelöli, hogy az adott elem vagy az első, vagy az utolsó. A lista elemek létrehozásáért a lista osztály felel, neki is kell beállítania a pointerait. Lista elemnek egyetlen konstruktora létezik, amely egy „T” generikus típust fogad el paraméterként, amire aztán inicializálja a megfelelő adatot, míg pointerait NULL-ra állítja.



1.Ábra Lista Elem  
UML-diagram



2. Ábra: Lista  
UML-diagram

##### 3.1.2. Lista

A lista lesz az a class, amit a felhasználók által is tudnak használni. A listából lehet objektumot készíteni és ezen az objektumon keresztül lehet majd új elemeket hozzáfűzni, meglévő elemeket törölni, teljes listát törölni és az egyéb műveleteket elvégezni.

A lista a char típusú stringeket automatikusan helyesen kezeli.

##### Lista által tárolt tagok:

A lista két „lista elemet” ér el közvetlenül. Az első és utolsó tagot. Ezeknek a mutatóit tárolja. (Azért van szükség mindkét tag tárolására, mert van lehetőség arra, hogy új elem felvételekor a lista elejére vagy végére fűzzük azt. Ezért szerencsétlen lenne mindig bejárni az egész listát, hogyha a „rossz” végére szeretnénk az új elemet felvenni.)

A lista továbbá tárol egy size\_t típusú értéket, ami a lista hosszát tárolja. Konstruktorkíváskor ez 0 kezdőértéket kapja.

##### Lista függvényei:

###### Default konstruktor:

*Minden pointert NULL-ra, hosszt 0-ra állít*

###### Copy konstruktor:

*Létrehoz egy másolatot a paraméterként átadott listáról.*

List(const T\* dataArray, size\_t sizeOfArray) [Konstruktor]:

*Létrehoz egy listát a paraméterben megadott tömbből*

Destruktor:

*Felszabadítja a listában tárolt lista elemeket.*

T& at(size\_t idx):

*Visszaadja az első elemtől „idx” távolságra tárolt „T” generikus típusú adattag referenciáját. (Ugyanez a függvény definiálva van konstans adattípusra is.) Ha az index értéke a listán kívülre mutat, kivételt dob.*

void push\_front(const T& data):

*A listában tárolt első elem elé helyezi el a paraméterként kapott új adattagot. (Ezzel az új adattag lesz az első elem.)*

void push\_back(const T& data):

*Ugyan az, mint a push\_front, csak a lista végére helyezi.*

void push\_at(const T& data, size\_t idx):

*Az első elemtől „idx” távolságra szúrja be az új adattagot. Ha az index értéke a lista lehetséges határain kívülre esne, kivételt dob.*

void clear():

*Végigfut a listaelemeken és törli azokat, majd a lista hosszát 0-ra állítja.*

void delete\_duplicates(Cmp cmp = Compare::DefaultEqual<T>()):

*Végigfut a lista elemein és ha egy elemből többet is talál, akkor törli azokat. (Egy példányt meghagy.)*

*[cmp functor ellenőrzi, hogy a két adat megegyezik-e]*

*[DefaultEqual: megegyezik az egyenlőség vizsgáló operátorral (operator==())]*

void sort(Cmp cmp = Compare::DefaultBigger):

*Végigfut a listán és rendezi az elemeit növekvő sorrendbe.*

*[cmp functor alapján végzi a rendezést]*

*[DefaultBigger: megegyezik a „nagyobb mint” operátorral (operator>())]*

size\_t get\_length() const:

*Visszatér a listahosszával.*

T& operator[](size\_t idx):

*Ugyan az, mint az at() függvény. (Ez is definiálva van konstans adattagokra is.)*

Iterator begin() const:

*Visszatér a Lista elejére mutató iterátorral.*

Iterator end() const:

*Visszatér a lista végét jelző iterátorral.*

### 3.1.3 Iterátor

Az iterátor egy olyan osztály, amit a lista namespace-en belül érhetünk el. Célja a lista bejárása és az adattagok elérése.

Egy adatot tárol, ami mindig az éppen elérni kívánt lista elem mutatója. Ez a mutató közvetlen nem érhető el, a tárolt elem elérése kizárólag a definiált függvényekkel érhető el.

#### Iterátor függvényei:

##### Default konstruktor:

*A tárolt mutatót NULL-ra állítja.*

##### Iterator(const List& l) [Ez is konstruktor]:

*A tárolt mutatót a paraméterként kapott lista első elemére állítja.*

##### Pre- növelő/csökkentő operátor:

*Előre/Hátra lépteti az iterátort és visszatér az új értéket tároló iterátorral.*

##### Post- növelő/csökkentő operátor:

*Előre/Hátra lépteti az iterátort és visszatér a léptetés előtti iterátor másolatával.*

##### bool operator!=(const Iterator& i) const:

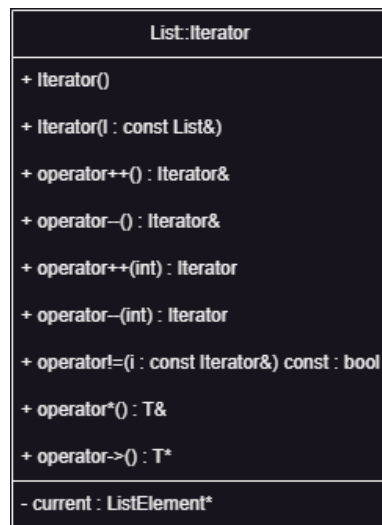
*Ellenőrzi, hogy a paraméterként kapott iterátorban tárolt mutató megegyezik-e saját maga által tárolt mutatóval. Ha nem akkor igazgal, egyéb esetben hamissal tár vissza.*

##### T& operator\*():

*Visszatér a mutatóban tárolt generikus adattag referenciájával. Ha a pointer NULL-ra mutat, akkor kivételt dob.*

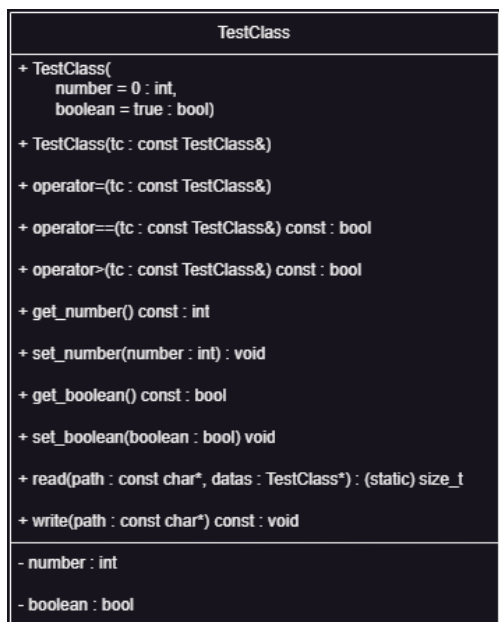
##### T\* operator->():

*Visszatér a mutatóban tárolt generikus adattag mutatójával. Ha a pointer NULL-ra mutat, akkor kivételt dob.*



3. Ábra: Iterátor UML-diagram

### 3.1.4. Teszt Class



A Teszt Class-nak a feladata, hogy különböző teszteseteken keresztül be lehessen mutatni a lista működését. Nincs egyéb funkcionalitása a program működését tekintve.

A Teszt Class továbbá képes io-műveleteket megvalósítani szöveges fájl formátumban.

#### Írás esetén:

Létrehoz egy fájlt, ha még nem létezett, vagy hozzáadja magát egy fájlhoz, ha az már létezett.

#### Olvasás esetén:

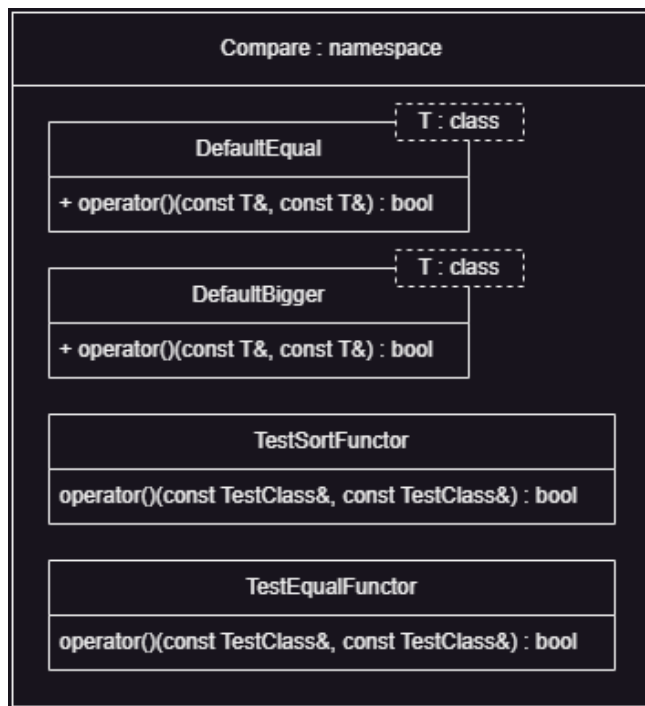
Ha a fájl megfelelően van szerkesztve, akkor a paraméterben megadott pointerre létrehoz egy tömböt, és visszatér a beolvasott fájlok mennyiségével.

(Hibásan szerkesztett fájl esetén kivételt dob)

4. Ábra: Teszt Osztály UML-diagram

### 3.1.5 Összehasonlító Functorok

A rendezéshez és a több megegyező példány törléséhez szükség van, hogy a lista elemeit összehasonlítsuk. Ehhez functorokat használhatunk. A házi feladatban 4 funcort készítettem el, amiből kettő a listához tartozik, általános összehasonlításokra (`operator==()`, `operator>()`), illetve ezek a `char` típusú stringekre specializálva), míg a másik kettő speciálisan a `TestClass` típusú objektumokat képes összehasonlítani.



### 3.1.6 Teljes UML-diagram





## 3.2 Algoritmusok

```
void List<T>::push_back(const T& newItem)
{
    ListElement* newElem = new ListElement(newItem);

    if (first == NULL)
    {
        first = newElem;
        last = first;
    }

    else
    {
        last->next = newElem;
        newElem->prev = last;

        last = newElem;
    }

    length++;
}

void List<T>::push_front(const T& newItem)
{
    if (first == NULL)
    {
        push_back(newItem);
        return;
    }

    ListElement* newElem = new ListElement(newItem);
    newElem->next = first;
    first->prev = newElem;

    first = newElem;
    length++;
}
```

```

void List<T>::push_at(const T& newItem, size_t idx)
{
    if(idx < 0 || idx > length)
        throw std::out_of_range("index can not be less than 0 or greater than the
current length");

    if(first == NULL || idx == length)
    {
        push_back(newItem);
        return;
    }

    if(idx == 0)
    {
        push_front(newItem);
        return;
    }

    ListElement* newElem = new ListElement(newItem);
    ListElement* prevNewElemIter = first;
    for (size_t i = 0; i < (idx - 1); i++)
    {
        prevNewElemIter = prevNewElemIter->next;
    }

    //Új elem beállítása
    newElem->prev = prevNewElemIter;
    newElem->next = prevNewElemIter->next;

    //Szomszédok beállítása:
    prevNewElemIter->next->prev = newElem;
    prevNewElemIter->next = newElem;

    length++;
}

void List<T>::clear()
{
    ListElement* iter = first;

    while (iter != NULL)
    {
        ListElement* tmp = iter;

        iter = iter->next;
        delete tmp;
    }

    first = NULL;
    last = NULL;
    length = 0;
}

```

```

void List<T>::swap(T& t1, T& t2)
{
    T tmp = t1;
    t1 = t2;
    t2 = tmp;
}

void List<T>::delete_duplicates(Cmp cmp)
{
    if (length <= 1) return; // Nem lehet benne duplikáció

    ListElement* outer_iter = first;
    size_t outIndex = 0;

    while (outer_iter != NULL)
    {
        size_t deleteIndex = outIndex + 1;
        ListElement* inner_iter = outer_iter->next;

        while (inner_iter != NULL)
        {
            ListElement* tmp = inner_iter;
            inner_iter = inner_iter->next;

            if (cmp(tmp->data, outer_iter->data))
            {
                delete_at(deleteIndex);
            }

            deleteIndex++;
        }

        outIndex++;
        outer_iter = outer_iter->next;
    }
}

void List<T>::sort(Cmp cmp)
{
    for (size_t i = 0; i < length-1; i++)
    {
        bool swapped = false;
        for (size_t j = 0; j < length - i - 1; j++)
        {
            if(cmp(at(j), at(j+1)))
            {
                swap(at(j), at(j+1));
                swapped = true;
            }
        }

        if (!swapped) break;
    }
}

```

```

T& List<T>::at(size_t index)
{
    if(first == NULL || length == 0)
        throw std::out_of_range("list has no members yet");
    if(index < 0 || index >= length)
        throw std::out_of_range("index can not be less than 0 or greater or equal to
the current length");

    ListElement* iter = first;
    for (size_t i = 0; i < index; i++)
    {
        iter = iter->next;
    }

    return iter->data;
}

List<T>::Iterator::Iterator() : current(NULL) {}
List<T>::Iterator::Iterator(const List& l) : current(l.first) {}
List<T>::Iterator& List<T>::Iterator::operator++()
{
    if(current == NULL) throw std::out_of_range("Iter run out of range");

    current = current->next;
    return *this;
}

List<T>::Iterator List<T>::Iterator::operator++(int)
{
    Iterator tmp = *this;
    operator++();
    return tmp;
}

T& List<T>::Iterator::operator*()
{
    if (current != NULL) return current->data;

    throw std::out_of_range("Iter pointing to NULL");
}

T* List<T>::Iterator::operator->()
{
    if (current != NULL) return &(current->data);

    throw std::out_of_range("Iter pointing to NULL");
}

bool Compare::DefaultBigger<T>::operator()(const T& t1, const T& t2)
{
    return t1 > t2;
}

```

```

inline bool Compare::DefaultBigger<const char*>::operator()(const char* c1, const char* c2)
{
    return strcmp(c1, c2) > 0;
}

bool Compare::DefaultEqual<T>::operator()(const T& t1, const T& t2)
{
    return t1 == t2;
}

inline bool Compare::DefaultBigger<char*>::operator()(char* c1, char* c2)
{
    return strcmp(c1, c2) > 0;
}

class TestSortFunctor
{
public:
    bool operator()(const TestClass& tc1, const TestClass& tc2);
};

class TestEqualFunctor
{
public:
    bool operator()(const TestClass& tc1, const TestClass& tc2);
};

```

## 4. Megvalósítás

A tervezett és a megvalósított lista között nincs eltérés, ezért a tervben megtalálható a használt osztályok publikus interfésze.

### 4.1 Tesztprogram bemutatása

A program futtatásakor egy menüben választhatjuk, hogy milyen teszteseteket akarunk futtatni. Van lehetőség minden tesztet egyszerre lefuttatni, de lehet csak egyes tesztcsoportokat is.

Továbbá van lehetőség arra is, hogy „Normál módon” futtassuk a programot, ilyenkor átlép egy alprogramba, ahol egész számokat lehet begépelni, amiket felvesz egy láncolt listába. Kilépéskor A láncolt lista tartalmát kiírja egy fájlba, majd a fájl tartalmát újra beolvassa egy újabb láncolt listába és végül kiírja az új lista tartalmát a konzolra.

### 4.2. Futtatott tesztesetek

A tesztek futtatásához a gtest\_lite könyvtárat használtam.

#### 4.2.1 Egész számokra futtatott tesztesetek. [void TestInt()]

Teszteli:

- Lista „int”-ekre való inicializálását
- Lista feltöltését elemekkel (előre, hátra, megadott helyre)
- Teljes lista törlését
- Iterátor működését
- Lista rendezését
- Lista elemeinek adott indexen való törlését
- Listában található duplikációk törlését

És a fent említett metódusok kivételdobásait is.

#### 4.2.2 CStringgeken futtatott tesztek

Teszteli:

- Lista const char\* stringekre való inicializálását
- Lista rendezését
- Lista duplikációinak törlését

#### 4.2.3 Saját osztályon futtatott tesztek

Teszteli:

- Lista feltöltését egy saját osztállyal és a listaelemek elérését az oparator[]-ral
- Lista elemeinek elérését iterátorral
- Lista rendezését összehasonlító funktorral
- Lista teljes tartalmának törlését
- Lista tartalmának írását és olvasását

És a fent említett metódusok kivételdobásait

#### 4.3 Memóriakezelés

A memóriaszivárgás teszteléséhez a memtrace modult használtam, ami nem mutatott hibát a végleges program futtatása során.