



Eötvös Loránd Tudományegyetem
Informatikai Kar

Alkalmazott modul: Programozás

11. fejezet

Strukturált programozás: dinamikus adatszerkezetek

Giachetta Roberto

A jegyzet az ELTE Informatikai Karának 2015. évi
Jegyzetpályázatának támogatásával készült

Dinamikus adatszerkezetek

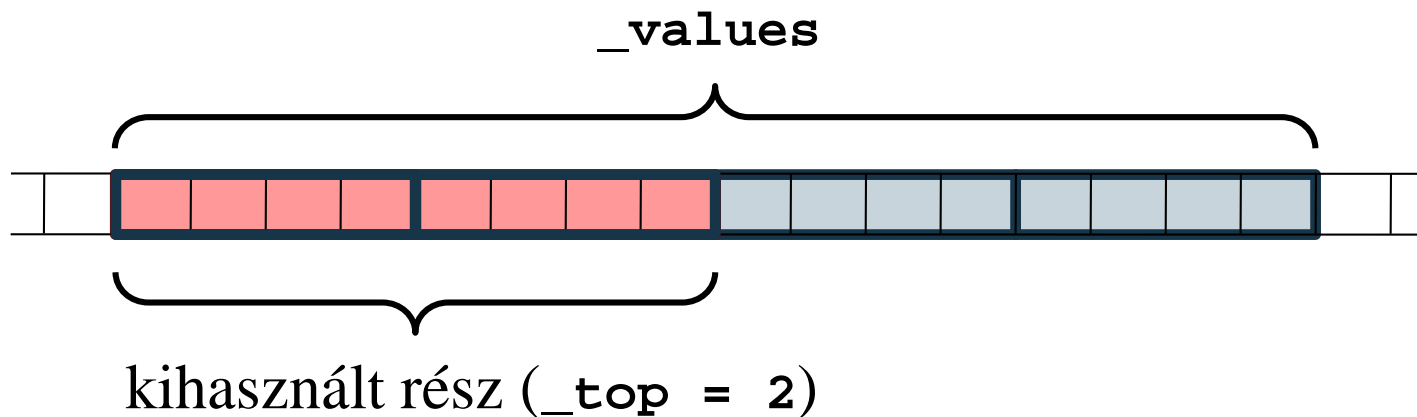
Dinamikus memóriakezelés

- A dinamikus memóriakezelés lehetővé teszi, hogy programfutás közben allokáljuk területeket a memóriából
 - a **new** és **delete** operátorok segítségével
 - több terület is foglalható egyszerre (dinamikus tömb)
 - a lefoglalt területeket mutatók segítségével kezeljük
- Saját típusokban is felhasználhatjuk a dinamikus memóriakezelés adta lehetőséget
 - elhelyezhetünk bennük mutatókat mezőként, amelyekre dinamikusan allokálhatunk memóriaterületet
 - az allokálást többször is elvégezhetjük a programfutás során

Dinamikus adatszerkezetek

Dinamikus méretezés

- A dinamikusan allokált tömb alkalmas arra, hogy futás közben szabályozzuk méretet
 - pl. verem esetén allokálnunk egy területet, amely egy részét fogja kitenni a verem tényleges tartalma



- probléma, hogy így akkor is nagy területet kell allokálni, ha jórészt kevésbé használjuk ki

Dinamikus adatszerkezetek

Dinamikus méretezés

- A dinamikusan lefoglalt terület nem méretezhető át, de lehetőségünk van új területet foglalni, és arra lecserélni a régit
- A következő lépéseket kell elvégeznünk:
 1. új tömb dinamikus lefoglalása
 2. az elemek átmásolása az új tömbbe
 3. a régi tömb törlése, lecserélése az újra (mutató átállítása)
- Ez jelentős műveletigényt jelent, ezért ritkán alkalmazzuk, nem egyesével növeljük a méretet, hanem rögtön duplájára
- Fordítva is alkalmazható, felezhetünk is méretet (így nem foglalunk felesleges memóriát)

Dinamikus adatszerkezetek

Dinamikus méretezés

- Pl.:

```
int* values = new int[10];  
    // 10 méretű dinamikus tömb  
... // 10 elemet behelyeztünk, szeretnénk 11-et is
```

```
int* t = new int[20]; // új tömb lefoglalása  
for (int i = 0; i < 10; i++)  
    t[i] = values[i]; // elemek áthelyezése
```

```
... // 11. elem behelyezése
```

```
delete[] values; // régi tömb törlése  
values = t;  
    // a mutató átállításával lecserélődik a tömb
```

Dinamikus adatszerkezetek

Példa

Feladat: Módosítsuk úgy a verem típust, hogy futás közben automatikusan átméreteződjön.

- ha megtelik, a duplájára méretezzük, ha negyedénél kevesebb eleme lesz, a felére méretezzük, ehhez felvesszük a `resize(int)` műveletet rejtett metódusként, ezt hívjuk meg a `push(T)` és `pop()` műveletekben
- az átméretezéskor létrehozunk egy új tömböt, a hasznos elemeket belehelyezzük (a `_top` indexig), majd lecseréljük a régi tömböt
- kezdetben a 100-as alapértelmezett méretet adjuk neki, ezért nem kell méretet adni a konstruktorban

Dinamikus adatszerkezetek

Példa

Megoldás:

```
template <class T> // elemtípus sablonja
class Stack { // verem típus
private:
    ...
    void resize(int newSize);
        // átméretezés megadott méretre
public:
    enum Exceptions { STACK_EMPTY };
        // kivételek felsorolási típusa

    Stack();
    ...
};
```

Dinamikus adatszerkezetek

Példa

Megoldás:

```
template <class T>
void Stack<T>::resize(int newSize) {
    T *temp = new T[newSize];
    // új tömb lefoglalása

    for (int i = 0; i < _top; i++){
        temp[i] = _values[i]; // értékek átmásolása
    }
    delete[] _values; // régi tömb törlése

    _values = temp; // új tömb beállítása
    _size = newSize; // eltároljuk az új méretet
}
```


Dinamikus adatszerkezetek

Példányok másolása

- Értékek másolásakor két változatot különböztetünk meg:
 - *mély másolat (deep copy)*: a teljes példány minden mezőjével lemásolódik a memóriában egy ugyanakkora memóriaterületre
 - lassú, mert sok adatot kell átmásolnunk
 - a másolaton végzett műveletek nem hatnak az eredetire
 - *sekély másolat (shallow copy)*: a példány nem, csak annak hivatkozása (memóriacíme) másolódik a memóriában
 - gyors, mivel csak egy adatot másolunk
 - a másolaton végzett műveletek kihatnak az eredetire
 - ehhez használjuk a referenciaorert, illetve mutatókat

Dinamikus adatszerkezetek

Példányok másolása

- Azonban dinamikus adatszerkezetek mély másolata problémás, mivel az alapértelmezett mély másolat a mezőkről készít másolatot, azonban *a mezőkre ráállított dinamikus területekről nem*, pl.:

```
Stack<int> s1;  
Stack<int> s2 = s1; // (elvileg mély) másolat  
s1.push(1);  
s2.push(2); // független elembehelyezések  
cout << s1.pop() << endl;  
    // eredménye: 2  
    // azaz az értékadások mégse voltak függetlenek
```

- Ahhoz, hogy ezt megoldjuk, felül kell definiálnunk az alapértelmezett másolást

Dinamikus adatszerkezetek

Másoló konstruktor

- A példányok másolását két művelet, a *másoló konstruktor* (*copy constructor*), illetve az *értékadás operátor* (=) valósítja meg
- A másoló konstruktor egy már létező példány alapján hoz létre újat
 - paraméterben megkapja egy ugyanolyan típusú példány referenciáját, törzsében elvégzi a másoló műveleteket
 - amennyiben nincs dinamikus tartalom a mezőkben, az alapértelmezett másoló konstruktor megfelelő
 - amennyiben van dinamikus tartalom, azt létre kell hozni, és az értékeket megfelelően belemásolni

Dinamikus adatszerkezetek

Másoló konstruktor

- Pl.:

```
class MyType {  
private:  
    int* _value;  
public:  
    MyType(int v = 0){ // konstruktor  
        _value = new int; *_value = v;  
    }  
    MyType(const MyType& other) { // másoló konstr.  
        _value = new int;  
        // a dinamikus tartalom létrehozása  
        *_value = *other._value; // érték másolása  
    }  
};
```

- A másoló konstruktor törzsében tud hivatkozni a másolandó példány mezőire
 - általában ezeket egyenként értékül adjuk az új objektumnak (összetett típusok esetén meghívódik azok másoló konstruktora)
 - természetesen további inicializálásokat is végezhetünk
- A másoló konstruktor a következő esetekben fut le:
 - közvetlen meghívás: `MyType b(a);`
 - kezdeti értékadás: `MyType b = a;`
 - érték szerinti paraméterátadás

Dinamikus adatszerkezetek

Példány hivatkozások

- Egy típuspéldányban elérhető annak valamennyi mezője és metódusa, akár rejtett, akár nem
- Ugyanakkor lehetőség van magát a teljes példányt is elérni a típuson belül a **this** kulcsszó használatával
 - ez egy mutatót ad vissza az aktuális példányra
 - ugyanúgy használható, mint bármely más mutató, amelyet az példányra állítottunk
 - lekérdezhető általa az összes tag: **this-><tagnév>**
 - lehetőséget ad a példány hivatkozásának visszaadására
 - ha a konkrét példányra van szükségünk, akkor lekérjük a memóriacím tartalmát (***this**)

Dinamikus adatszerkezetek

Példány hivatkozások

- Pl.:

```
class MyType {  
public:  
    ...  
    void SetValue(int value){  
        *_value = value;  
        // ugyanez hosszabban:  
        // *(this->_value) = value;  
    }  
    MyType* ReturnMyPointer(){  
        return this;  
        // visszaadja a mutatót a példányra  
    }  
}
```

Dinamikus adatszerkezetek

Példány hivatkozások

- Pl.:

```
    MyType ReturnMyself () {  
        return *this;  
        // visszaadja magát a példányt  
    }  
private:  
    int *_value;  
};  
...  
MyType t;  
MyType* tP = t.ReturnMyPointer();  
tP->SetValue(10);  
    // másként: t.SetValue(10);
```


Dinamikus adatszerkezetek

Értékadás operátor

- Amennyiben a változó értékét a későbbiekben módosítjuk, az értékadás operátor fut le:
`MyType a, b;`
`b = a; // értékadás`
- Az értékadás operátor megkapja a másolandó példány (konstans) referenciáját, és biztosítja tartalmának átmásolását
 - fontos, hogy már léteznek a dinamikusan létrehozott értékek, így azokat törölni kell
 - ellenőrizni kell, hogy a paraméterben kapott változó nem-e saját maga (a memóriacím segítségével)
 - a többszörös értékadás használatához vissza kell adnia az aktuális példány (`*this`) referenciáját

Dinamikus adatszerkezetek

Értékadás operátor

- Pl.:

```
class MyType {  
public:  
    ...  
    MyType& operator=(const MyType& other){  
        if (this == &other)  
            // ha ugyanazt a példányt kaptuk  
            return *this; // nem csinálunk semmit  
  
        *_value = *(other._value);  
        // különben a megfelelő módon másolunk  
        return *this; // visszaadjuk a referenciát  
    }  
};
```

Dinamikus adatszerkezetek

Dinamikus típusok műveletei

- Amennyiben dinamikusan lefoglalt memóriaterületet használunk a típusban, minden esetben meg kell valósítani a destruktort, a másoló konstruktort, és az értékadás operátort
 - ha nincs dinamikus tartalom, akkor is előfordulhat, hogy használjuk őket
 - mindhárom művelet csak metódusként írható meg
 - mindhárom műveletnek publikusnak kell lennie
 - az értékadás operátor teljesen független az értékmódosító operátoroktól ($+=$, $*=$, ...), amelyek megvalósíthatók a típuson kívül is

Dinamikus adatszerkezetek

Példa

Feladat: Módosítsuk a verem típus úgy, hogy lehessen megfelelő mély másolatot készíteni.

- megvalósítjuk a másoló konstruktort és az értékadás operátort

Megoldás:

```
template <class T> // elemtípus sablonja
class Stack { // verem típus
private:
    ...
    Stack(const Stack<T>& other);
    ...
    Stack<T>& operator=(const Stack<T>& other);
```

Dinamikus adatszerkezetek

Példa

Megoldás:

...

```
template <class T>
```

```
Stack<T>::Stack(const Stack& other) {
```

```
    _size = other._size;
```

```
    // átmásoljuk az egyszerű értékeket
```

```
    _top = other._top;
```

```
    _values = new T[_size];
```

```
    // létrehozuk a dinamikus tömböt
```

```
    for (int i = 0; i < _top; i++)
```

```
        _values[i] = other._values[i];
```

```
    // átmásoljuk a hasznos értékeket
```

```
}
```

Dinamikus adatszerkezetek

Példa

Megoldás:

```
template <class T>
Stack<T>& Stack<T>::operator=(const Stack<T>&
                               other) {

    if (this == &other)
        // amennyiben a két memóriacím megegyezik
        return *this; // nem kell semmit csinálni

    delete[] _values;
    // eddig használt dinamikus tömb törlése
    ...
    return *this;
}
```

- Saját típusainkhoz lehetőségünk van indexelő (`[]`) operátort készíteni
 - az indexelő operátornak egy tetszőleges típusú értéket adhatunk meg (az operátoron belül), ez kerül át a paraméterbe
 - amennyiben több értékkel szeretnénk indexelni (pl. mátrix esetén sor/oszlop), használhatjuk a funktor (`()`) operátort, amely zárójelezéssel hívható meg, tetszőleges sok paraméter adható át
 - indexelés esetén külön kell ügyelni a beállítás, illetve a lekérdezés kérdésére, ezért az operátort túl kell terhelni (a túlterhelést a **const** kulcsszó fogja biztosítani)

Dinamikus adatszerkezetek

Indexelés

- Pl.:

```
class MyType {  
private:  
    int _values[100]; // értékek tömbje  
public:  
    int operator[](int index) const {  
        // indexelő operátor lekérdezéshez  
        return _values[index]; // értéket ad vissza  
    }  
    int& operator[](int index) {  
        // indexelő operátor értékadáshoz  
        return _values[index];  
        // referenciát ad vissza  
    }  
    ...  
}
```


Dinamikus adatszerkezetek

Példa

Feladat: Valósítsuk meg az intelligens dinamikus vektor (**vector**) típust, amelyet futás közben lehet bővíteni.

- a vektor sablonos lesz, primitív dinamikus tömbbel ábrázoljuk, amely automatikusan méreteződik
- mindig egy részét használjuk ki a teljes tömb kapacitásának (hasonlóan, mint a verem esetében), a konstruktor paraméterben kapja meg a kezdeti méretet
- a **push_back(T)** művelettel bővíthető, a **pop_back()** művelettel redukálható, a **clear()** művelettel üríthető, méretét a **size()** művelettel kérdezhetjük le
- az elemek elérését/beállítását az indexelő operátor biztosítja

Dinamikus adatszerkezetek

Példa

Megoldás:

```
template <class T>
class vector { // dinamikus vektor típusa
public:
    vector(int size = 0); // konstruktor
    vector(const vector<T>& other);
        // másoló konstruktor
    ~vector(); // destruktork
    void push_back(T value);
        // elem beszúrása a vektor végébe
    T pop_back(); // utolsó elem törlése
    int size() const; // méretlekérdezés
    void clear(); // kiürítés
```

Dinamikus adatszerkezetek

Példa

Megoldás:

```
T& operator[](int index);  
    // indexelő operátor beírásra  
T operator[](int index) const;  
    // indexelő operátor lekérdezésre  
  
vector<T>& operator=(const vector<T>& other);  
    // értékadás  
  
enum Exceptions { SIZE_INVALID, VECTOR_EMPTY,  
    INDEX_OUT_OF_RANGE }; // kivételek  
private:  
    void resize(int newCapacity);  
    // tömb átméretezése
```

Dinamikus adatszerkezetek

Példa

Megoldás:

```
T* _values; // értékek tömbjének mutatója
int _size; // vektor aktuális mérete
int _capacity;
    // vektor kapacitása (teljes mérete)
};
...
template <class T>
T& vector<T>::operator[](int index) {
    if (index < 0 || index >= _size)
        // ellenőrizzük az indexet
        throw INDEX_OUT_OF_RANGE;
    return _values[index];
}
```

Dinamikus adatszerkezetek

Típusok dinamikus kezelése

- Saját típusainkat is létrehozhatjuk, illetve törölhetjük dinamikusán:

```
<típusnév> * <mutatónév> = new <típusnév>;  
delete <mutatónév>;
```

- amennyiben a saját típusunk konstruktorparaméterekkel rendelkezik, azokat meg kell adnunk a létrehozáskor:
<mutatónév> = new <típusnév>(<paraméterek>);
- Továbbra is lehetőségünk van hivatkozni a típusunk adattagjaira (** <mutatónév> . <tagnév>* formában
 - a zárójel az operátor precedencia miatt kell
 - mivel ez elég összetett jelölés, lehet egyszerűsíteni a -> operátorral: *<mutatónév>-><tagnév>*

Dinamikus adatszerkezetek

Típusok dinamikus kezelése

- Pl.:

```
struct MyType {  
    int value;  
    void print() { cout << Value; }  
    MyType() { Value = 0; } // konstruktorok  
    MyType(int v) { Value = v; }  
};  
...  
MyType *d1, *d2; // mutató létrehozása  
d1 = new MyType; // példányosítás konstruktorral  
d1->print(); // 0, ugyanez: (*d1).Print()  
d2 = new MyType(10); // paraméteres konstruktorral  
d2->print(); // 10  
delete d1; delete d2; // törlések
```

- A típuson belül lehetőségünk a típusnak a mutatóját, azaz egy *reflexív mutatót* mezőként elhelyezni, pl.:

```
struct MyType {  
    MyType* myPointer;  
    // mutató ugyanarra a típusra  
    ...  
}  
MyType mt;  
mt.myPointer = new MyType;  
    // a mutatóra ráhelyezhetünk egy új példányt  
mt.myPointer->myPointer = new MyType;  
    // annak a mutatójára is ráhelyezhetünk egy  
    // új példányt
```

Dinamikus adatszerkezetek

Láncolás

- A reflexív mutatók segítségével összeállított elemek sorozatát nevezzük *láncolt sorozat*nak, ha pedig adatszerkezet elemeit valósítjuk meg így, akkor a struktúrát *láncolt adatszerkezet*nek
 - a láncolt adatszerkezet elemei olyan rekordok, amelyek tartalmazznak legalább egy mutatót a saját típusra, e mentén láncolhatóak
 - a láncolás előnye, hogy a láncba könnyű beszúrni/kitörölni elemet, tehát tetszőleges ponton módosítható a lánc
 - a láncolt adatszerkezet pontosan annyi elemet tárol, mint amennyit ténylegesen hasznosítunk, így nincs felesleges memória foglalkozás (a rekordban lévő mutatókon kívül)

Dinamikus adatszerkezetek

Láncolás

- Pl.:

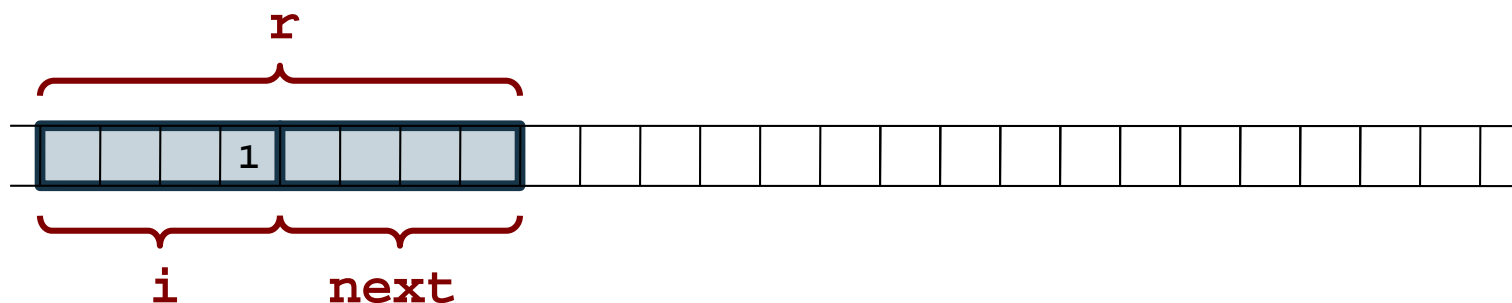
```
struct MyRecord { int value; MyRecord* next };
```

...

```
MyRecord * r = new MyRecord;
```

```
r->value = 1; // mezők beállítása
```

```
r->next = NULL;
```



Dinamikus adatszerkezetek

Láncolás

- Pl.:

```
struct MyRecord { int value; MyRecord* next };
```

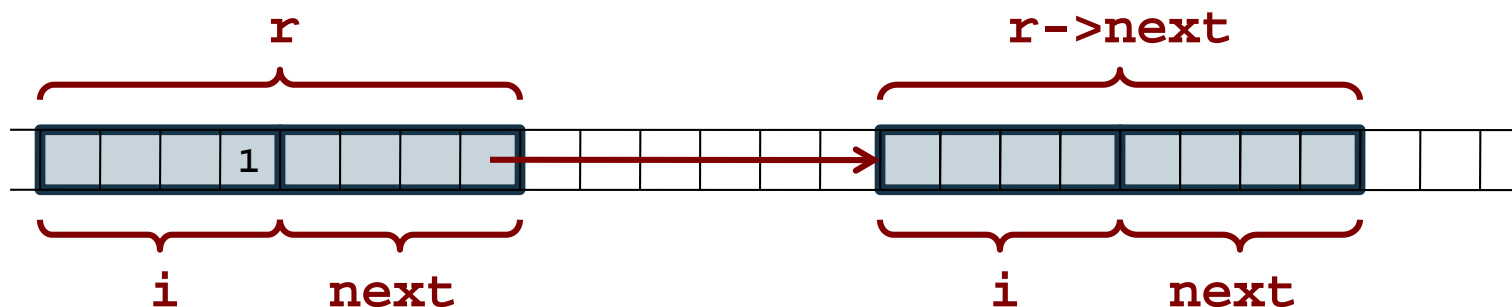
...

```
MyRecord * r = new MyRecord;
```

```
r->value = 1; // mezők beállítása
```

```
r->next = NULL;
```

```
r->next = new MyRecord; // új rekord
```



Dinamikus adatszerkezetek

Láncolás

- Pl.:

```
struct MyRecord { int value; MyRecord* next };
```

...

```
MyRecord * r = new MyRecord;
```

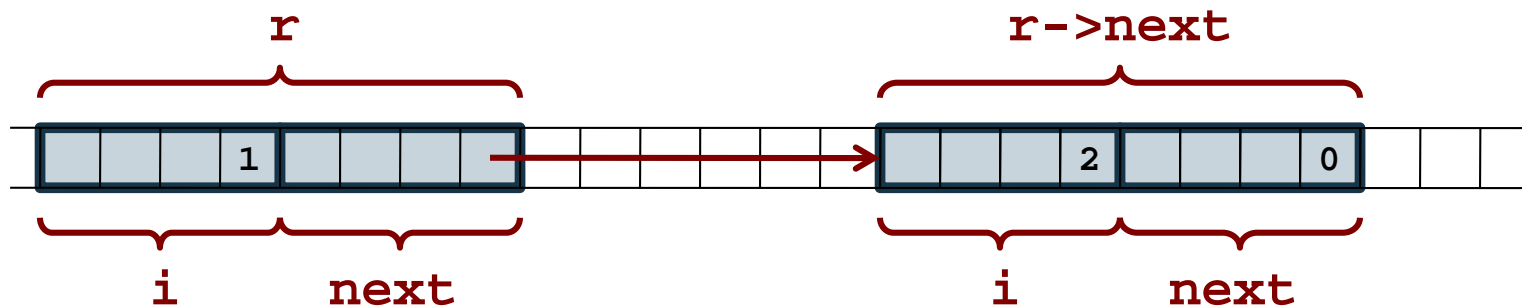
```
r->value = 1; // mezők beállítása
```

```
r->next = NULL;
```

```
r->next = new MyRecord; // új rekord
```

```
r->next->i = 2; // új rekord mezői
```

```
r->next->next = NULL;
```



Dinamikus adatszerkezetek

Láncolás bejárása

- A láncolt adatszerkezet bejárása (pl. másolásnál) nem történhet indexeléssel, hiszen a memóriában tetszőleges helyen helyezkedhetnek el az elemek, ezért mutatót kell használnunk
 - a mutatót beállítjuk az első elemre
 - addig haladunk, amíg `NULL` értékre nem lépünk
 - a következő elem mutatóját használjuk a lépéshez
 - pl.:

```
for (MyRecord* p = r; p != NULL; p = p->next){  
    // a p mutatóval lépkedünk előre az r által  
    // mutatott láncon  
    cout << p->value << endl;  
}
```

Dinamikus adatszerkezetek

Láncolt adatszerkezetek

- Láncolt adatszerkezetek esetén létre kell hozni az elemtípust is
 - általában egyszerű rekord (**struct**)
 - beágyazva deklaráljuk, azaz a típuson belül, pontosabban a rejtett elemek között (így csak a típuson belül lesz használható)
- A láncolt adatszerkezet mindig csak mutatót tárol, erre hozzuk létre dinamikusan az elemeket
 - másolásnál egyenként kell átmásolni az értékeket az újonnan lefoglalt elemekbe
 - törlés esetén ügyelni kell, hogy minden elemet megsemmisítsünk

Dinamikus adatszerkezetek

Verem láncolt megvalósítása

- A *verem* adatszerkezetet is meg lehet valósítani láncolással, itt a verem mutatója a legfelső elemre (**top**) mutat az adatszerkezetben, és azt követik a továbbiak
 - ha fel szeretnénk venni egy új elemet (**push**), akkor a többi elem elé kell tennünk a láncban, azaz létrehozuk az új rekordot, ráfűzzük a többit, és ráállítjuk a verem mutatóját
 - a verem mutatóján át hivatkozhatunk a tetőelemre (**top**)
 - ha ki szeretnénk venni a tetőelemet (**pop**), akkor egy segédváltozóval ki kell emelnünk az értéket, majd a veremből ki kell venni az első rekordot
 - ehhez újabb segédváltozó kell, amit törölünk, miután a veremmutatót átállítottuk

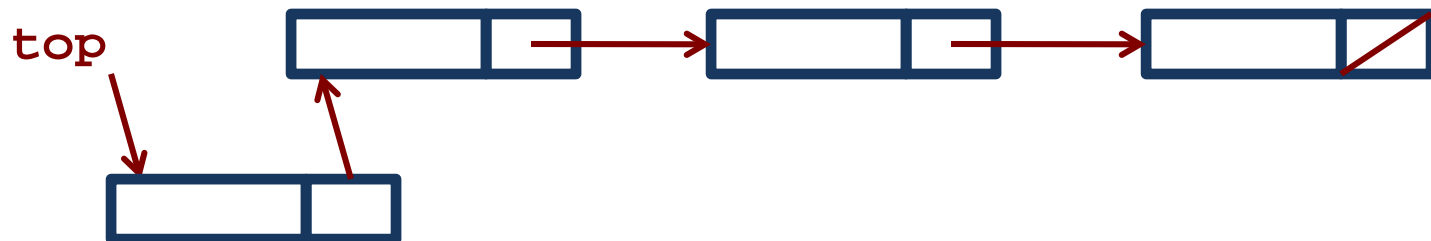
Dinamikus adatszerkezetek

Verem láncolt megvalósítása

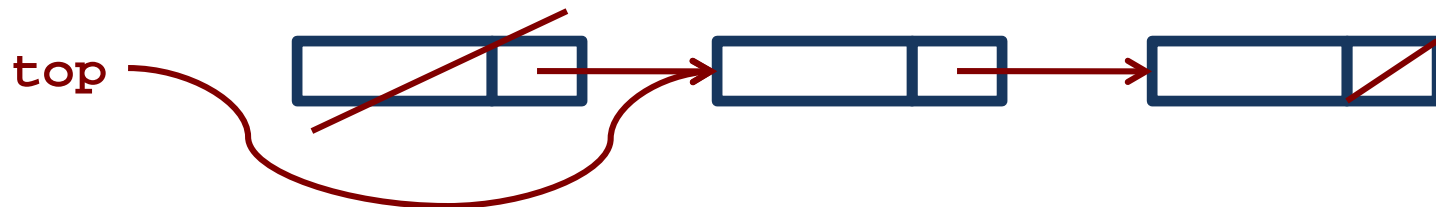
- A verem reprezentációja:



- Beszúrás a verembe:



- Kivétel a veremből:



Dinamikus adatszerkezetek

Példa

Feladat: Valósítsuk meg a verem adatszerkezetet láncolt reprezentációval.

- megvalósítjuk a veremelem típust (**StackItem**), amely tárolja az adatot és a következő elem mutatóját
- a típusban csak a tetőelem mutatóját (**_top**) és a méretet (**_size**) tároljuk el, utóbbi csak a méretlekérdezéshez szükséges
- beszúrásnál dinamikusán létrehozzuk az elemet, kivételnél töröljük
- a kiürítés (és a destruktor) voltaképpen a törlés alkalmazása az összes elemre

Dinamikus adatszerkezetek

Példa

Megoldás:

```
template <class T> // elemtípus sablonja
class Stack { // verem típus
private:
    struct StackItem {
        // veremelem típusa, mint beágyazott típus
        T _value; // a tárolt érték
        StackItem* next; // következő elem mutatója
    };

    StackItem* _top; // a tetőelem mutatója
    int _size; // méret (csak a lekérdezéshez kell)
    ...
};
```

Dinamikus adatszerkezetek

Példa

Megoldás:

```
template <class T>
void Stack<T>::push(T value) {
    StackItem* item = new StackItem;
    // létrehozzuk az új elemet
    item->_value = value; // beállítjuk az értékét
    item->next = _top;
    // ráhelyezzük a lánc eddigi részét
    _top = item; // berakjuk az elejére
    _size++;
}
```

Dinamikus adatszerkezetek

Példa

Megoldás:

```
template <class T>
T Stack<T>::pop() {
    if (_size > 0) { // ha van elem
        T data = _top->_value;
        // lekérdezzük az elemet
        StackItem* temp = _top;
        // segédmutató (a törléshez)
        _top = _top->next;
        // átállítjuk az első elemet
        delete temp; // kitöröljük az elemet
        _size--;
        return data; // visszaadjuk az értéket
    }
    ...
}
```

Dinamikus adatszerkezetek

Reprezentációja

- Az aritmetikai (tömbös) reprezentáció előnyei:
 - az elemek egymás után helyezkednek el, ezért indexelhetőek
 - nincs szükség további mutató tárolására az elemek behivatkozására
 - a műveletek könnyen megfogalmazhatóak
- A láncolt ábrázolás előnyei:
 - mindig annyi helyfoglalás történik, ahány elem van
 - tetszőleges ponton bővíthető/módosítható az adatszerkezet
 - nincs költsége az átméretezésnek
 - a láncolás több irányba is történhet