Első beadandó

Október 2019

Bevezető

Az első feladat célja egy olyan adatszerkezet megírása, amely tartalmazz egy előre lefoglalt fix méretű tömböt. Az adatszerkezet feladata az, hogy képes legyen ezzel a tömbbel gazdálkodni úgy, hogy a felhasználó képes legyen tőle memóriát kérni, ami a felhasználóé addig, amíg azt ő vissza nem adja. Ez azt jelenti, hogy senki másnak nem szabad odaadnia azt a memóriát és maga az adatszerkezet sem használhatja azt. A második feladat célja, hogy az előző adatszerkezetet felhasználva, azokból egy láncolt listát csinálva, dinamikus méretűvé változtatni ezt az adatszerkezetet.

1. Fix méretű memóriaterület (70p)

A feladat első eleme egy olyan adatszerkezet írása, amely összmérete és a blokk mérete is adott inicializáláskor.

1.1. Inicializálás

- Két template paramétert kell tudjunk megadni, az első a tárolt adat típusa (későbbiekben T), a második pedig hány darab ilyen adatot (blokkot) tudunk eltárolni maximum (N).
- A konstruktorban lefoglaljuk az összes szükséges memóriát, utána több memória foglalás nincsen (tehát gyakorlatilag egy new hívás van).

1.2. Működésbeli megkötések

Az allokátoroknak memória hatékonynak és gyorsnak kell lenniük ezért:

- Az osztályban (tehát nem külön lefoglalt területen), el lehet tárolni általános statisztikákat, pl. mennyi szabad, foglalt hely van, mekkora egy blokk... (max 6 integer, de tapasztalat szerint 4 tökeletesen elég) Valamint tartalmazhat egy pointert a lefoglalt memóriacímre, valamint egy másik szabadon felhasználható pointert. Ezen felül minden további memória használata módosítja a pontozást:

  - Ha legalább olyan hosszú (elemszámú) tömböt, kell lefoglalni, mint maga a lefoglalt memóriacím. **0p**
  - Ha csak azt tartja számon, hogy melyik területek szabadultak fel, tehát lineárisan nő ezzel arányosan. (Határhelyzetekben a javító belátására bízzuk) **10p**
  - Ha nincs szükség további memória felhasználására (hint: a szabad memória területeket, fel lehet használni a belső szerkezet leírására) Fel lehet azt használni, hogy egy blokk mérete legalább akkora, mint egy integer. **10 + 10p**

- A szabad memória megtalálása O(1)-nek kell lennie, tehát iterációk nélkül képesnek kell lennie mindig tudnia, hol van szabad hely. Ha ez nem teljesül, akkor a fenti memóriahatékonyság részre **-10p** jár, mert hiába foglal kevés memóriát, ha közben nem képes ezt gyors keresésre használni.

- *A bónusz ponthoz többféleképpen kell felhasználni egy adott memória területet. Viszont az alapból T típusú, de ha fel akarod használni más célra, mondjuk egy integert akarsz benne tárolni, amíg az szabad, akkor az úgynevezett reinterpret_cast hasznos lehet:*

```
T* p = mutat valahova a pool-ba;
int* k = reinterpret_cast<int*>(p);
*k = 5;
```

1.3. Kötelező függvények

Néhány példa az utolsó oldalon található.

- **allocate():** lefoglal egy blokk memória területet és visszadja az arra mutató pointert. Valamint std::bad_alloc exception-t, dob ha már nincs több hely. **15p**

- **allocate(unsigned int n):** Lefoglal n darab folytonos blokkot és visszadja az arra mutató pointert. Könnyítés képpen ezt csak akkor kell tudni teljesíteni, ha van még annyi olyan szabad hely, ami még soha nem volt lefoglalva. Tehát, akkor is lehet std::bad_alloc-t dobni, ha már túl sok memória le lett foglalva, de már felszabadult elég folytonos memória. A kötelező függvényeken kívűl, tartalmazhat más függvényeket is az adatszerkezet. **10p**

- **deallocate(T* p):** Felszabadítja a p által mutatott blokkot. Nem kell ellenőrizni, hogy az ténylegesen le van-e foglalva, ez a felhasználó felelőssége. A felszabadítás, alatt csak arra kell gondolni, hogy a benne levő adatot szabadon újra lehet használni, tehát a jövőben újra lefoglalhatónak kell lennie. **15p**

- **deallocate(T* p, unsigned int n):** Felszabadítja a p által mutatott n blokkot. Itt sem kell ellenőrizni, hogy foglalt-e. Ez lehet O(n). **10p**
- **construct(T value):** Ugyanúgy lefoglalja és visszadja a memória címet, de be is másolja oda a value-t. **10p**

Természetesen a kötelező függvényeken kívűl, tartalmazhat más függvényeket is az adatszerkezet.

2. Dinamikus allokátor 30p

Egyértelműen látszik az előző módszer nagy hiányossága, hogy ha kifogyunk a memóriából, akkor már csak exception-t kaphatunk, amíg nem szabadítunk föl új területet. Erre a megoldás a következő:

- Írj egy másik adatszerkezetet, amely felhasználja az előzőleg megírt fix méretű adatszerkezetet. Amikor az betelik akkor készít egy újat és ezeket összeláncolja Node-ok segítségével.
- Ennek is ugyanaz a két template paramétere kell, hogy legyen, de a második (N), azt mondja meg, hogy egyszerre maximum mekkora memórát képes lefoglalni.
- Új Node-t csak akkor lehet készíteni, ha egyik korábbiba se fér már bele további adat. Tehát ha közben felszabadult hely a korai Node-ban, akkor a következő memória kérésnél (allocate), akkor azt a területet is figyelembe kell venni a jövőben. **5p**
- Ennek az adatszerkezetnek is rendelkeznie kell a fent említett 5 kötelező függvénnyel (allocate(), allocate(unsigned), deallocate(T*), deallocate(T*,unsigned), construct(T)). **4+4+4+4+4p**
  - Itt a sima allocate() soha nem dobhat bad_allocation exception-t, mivel mindig képes új memóriát lefoglalni, új Node létrehozásával.
  - Az allocate(unsigned n), akkor dobhat bad_allocation exception-t, ha a kért n > N, tehát nem lenne képes egy Node-ban eltárolni. (Nem lenne folytonos a lefoglalt memória)
- Itt is van memória korlátozás. Maga az adatszerkezet legfejlebb 2 Node pointert tartalmazhat, valamint a másik esetben is specifikált max 6 integer általános statisztikát lehet letárolni. A Node-ok is legfeljebb 2 pointert tartalmazhatnak darabonként. **5p**

Megjegyzés: A fent említett memória korlátozások nagyon kényelmesen elegek, ebből jóval kevesebből is meg lehet oldani jól.

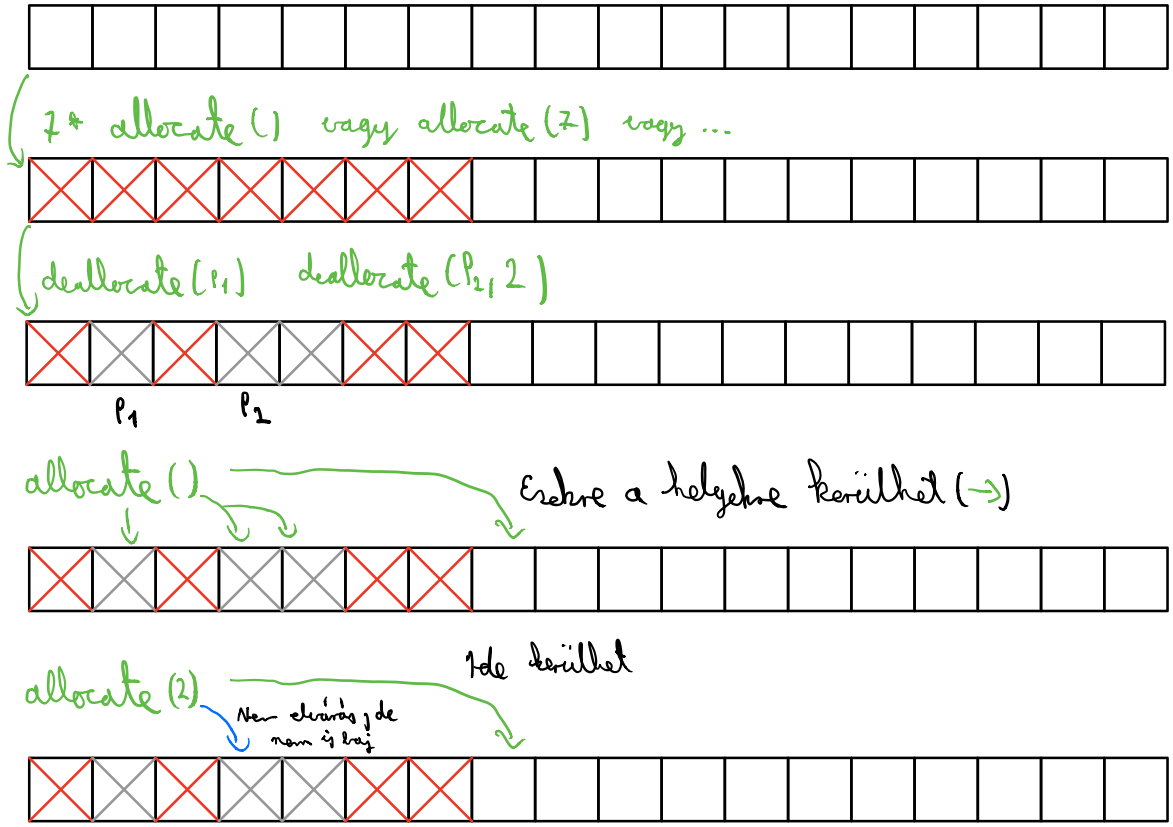
Megjegyzés: A deallocate függvényeknél felmerülhet, hogy mi történik, ha ott egy pointert is tartalmazó objektum van, aminek következtében esetleg a heapre létrehozott adatot elveszítjük. Ez valós veszély, de feltételezhetjük, hogy a T

az mindig egy olyan típus, amely csak helyben létezik és nem csinál semmiféle dinamikus memória kezelést. (Ha ezt ki akarnánk javítani, akkor destruktort kéne meghívni, de azt csak erősen indokolt esetben szabad, általában tilos)

3. Bónusz feladat (20 p)

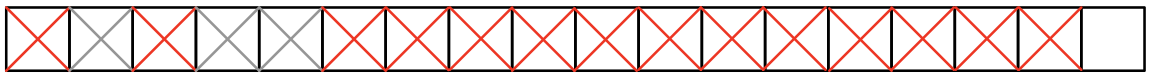
Az eddig leírt adatszerkezetek használhatóak allokátorként. Az allokátorok arra vannak, hogy a felhasználó számára memóriát adjon és ha már nincs szüksége rá, akkor újrahasznosítsa. Az alapértelmezett allocator az std::allocator, amikor létrehozunk egy vector-t vagy listát, akkor ezt az allokátort fogja használni, viszont lehetőségünk van saját allokátort használni (pl. *std::vector<int, sajat_allokator>*), az alapértelmezett helyett. Annyi az egyetlen megkötés, hogy nem lehet az std::allocator-ból leörökölni. Néhány segítség ehhez:

- Meg kell adni az úgynevezett allocator_traits-eket, ehhez kell egy kis kutató munka. Az alábbi linken fel vannak sorolva milyen típusokat kell definiálni (public-ként), például így: *using pointer = T*;*
- rebind-ra is szükség van, ehhez, ami egy olyan template-es sub-struct amiben egyetlen egy typedef van a sa ját allocatorod új template-es változatára. Interneten végtelen sok példa van erre.



---

allocate(2)



ide elfér, de nem elvárás, hogy ezt visszaadd

ide már nem fér el

bad_alloc és jó megoldás