Segédlet a „Programozási nyelvek - C++" című tantárgyhoz

Májer Viktor

December 20, 2009

Ez a jegyzet egy összefoglaló az STL konténereiről. Nem hivatalos referencia, csupán egy összefoglaló az interneten fellelhető referenciákból. Egyikük-másikuk régebbi, vagy kevésbé ellenőrzött, ezért szerepelhetnek pontatlanságok. Sok helyen az adatszerkezeteknek, függvényeknek pontosabb paraméterezései is adhatóak, de itt most csak az összefoglalást tartom lényegesnek.

Egyéb tudnivalókért, példákért keressétek fel a http://people.inf.elte.hu/majer_v/assist.html címen található linkeket.

Ha másolási hibát találtok, kérlek jelezzétek a jokmokk@gmail.com címen (elválasztási hibákat ne, azt a LyX töredeli így :))

A jegyzet folyamatos frissülése várható (hamarosan a fontosabb algoritmusokat, példákat, és a zh-n is hasznos egyéb fontos tudnivalókat is tartalmazni fogja), így mindig frissítsd onnan, ahonnan letöltötted. Addig is nézzétek át ezt. :)

Contents

rész I
Kivonat az STL-ből

A Standard Template Library (Szabványos Sablon Könyvtár - röviden STL) egy szoftver könyvtár, ami a C++ Szabványos Könyvtárának szerves részét képezi. Az STL konténerek (tárolók), algoritmusok és iterátorok gyűjteménye - számos informatikai alap algoritmust és adatszerkezetet tartalmaz. Az STL egy generikus könyvtár, vagyis az elemei erősen paraméterezhetőek, más szóval, C++-os terminológiával élve: a könyvtár majdnem minden eleme template. Ezért használatához erősen ajánlott a C++ template-ek működésének megértése.

1. Konténerek (tárolók)

Az STL egyik fontos részét képezik a tárolók, azok az adatszerkezetek, amelyek különféle tárolási stratégiákat implementálva hatékonyan, biztonságosan, kivételbiztosan és típushelyesen képesek tárolni az adatokat, ellentétben a C-stílusú, beépített tömbökkel és kézzel írt láncolt adatszerkezetekkel. Az STL konténer osztályai bármilyen típus tárolására példányosíthatók. A tároló osztályok a következők:

1. Szekvenciák: vector, deque, list, slist, bit vector
2. Asszociatív konténerek: set, map, multiset, multimap, hash set, hash map, hash multiset, hash multimap, hash
3. Konténer adapterek: stack, queue, priority queue

1.1. Szekvenciák

A szekvenciák változó méretű konténerek, melyek elemei (szigorúan) meghatározott lineáris sorrendben követik egymást. Támogatják az elemek beszúrását és törlését.

1.1.1. Vector

A vector voltaképp egy dinamikus tömb. Tetszőleges eleméhez támogatja a hozzáférést, konstans idejű beszúrást és törlést biztosít a végén, és lineáris idejűt az elején vagy a közepén. A vectorban tárolt elemek száma változó, a memóriakezelés automatikus. A vector a(z egyik) legegyszerűbb STL tároló, és sok esetben a leghatékonyabb is.
Használat: <**vector**> fejállományban
template < **class T, class Allocator** = **allocator**<**T**> > **class vector;**
ahol T a tárolt elemek típusa, Allocator pedig azok foglalásának módja

Tagfüggvények:

Konstruktorok:

1. **vector**();
2. **vector**(const vector& c);
3. **vector**(size_type num, const TYPE& val = TYPE());
4. **vector(** input_iterator start, input_iterator end);

5. **~vector**();

   Leírás:

   1. A default ctor: vigyázzunk arra, hogy ekkor a vector még üres (ne indexeljük!!)
   2. A copy ctor
   3. Megadhatunk egy méretet, mennyi elemnek foglaljon a vector helyet, és egy adott tárolt értéket (egyébként a tárolt típus default ctor()-val hozzuk létre az elemeket)
   4. Két iterátort váró konstruktor: a két iterátor által kijelölt intervallum elemeivel hozzuk létre az új vektort
   5. Destruktor

Operátorok:

1. TYPE& **operator[]**(size_type index);
2. const TYPE& **operator[]**(size_type index) const;
3. vector **operator**=(const vector& c2);
4. bool **operator**==(const vector& c1, const vector& c2);
5. bool **operator!**=(const vector& c1, const vector& c2);
6. bool **operator**<(const vector& c1, const vector& c2);
7. bool **operator**>(const vector& c1, const vector& c2);
8. bool **operator**<=(const vector& c1, const vector& c2);
9. bool **operator**>=(const vector& c1, const vector& c2);

   Leírás:

   Ami innen különösen fontos, az az 1-2 indexelő operátorok (figyelem, nem végez tartományellenőrzést!!), illetve a 3. értékadás operátor (hogy helyesen meg van írva)

   Többihez: két vector akkor számít egyenlőnek, ha elemszámuk és elemeik páronként (==) megegyeznek.

   (Természetesen a (const vector& c1, const vector& c2) paraméterezésű operátorok globálisak, nem tagok.)

Egyéb tagok:

- bool **empty**() const;

  visszaadja üres-e a vector

- size_type **size**() const;

  tárolt elemek száma

Elem lekérdezések:

- TYPE& **at**(size_type loc);

  elemlekérdezés index alapján (referenciát ad vissza, így írható is), de ez már végez tartományellenőrzést

- const TYPE& **at**(size_type loc) const;

  ugyanaz, csak const módon

- TYPE& **front**(); const TYPE& **front**() const;

  legelső elem lekérdezése const és nem const módon (az első felülírást is lehetővé tesz pl.: v.front()=7 adott esetben)

- TYPE& **back**(); const TYPE& **back**() const;

  legutolsó elem lekérdezése const és nem const módon (az első felülírást is lehetővé tesz pl.: v.back()=4 adott esetben)

Beszúrások:

- void **assign(** size_type num, const TYPE& val);

  hozzáad num darab val értékű tárolt elemet a vector-hoz

- void **assign**(input_iterator start, input_iterator end);

  az iterátorok által kijelölt tartomány elemeit adja hozzá

- iterator **insert**(iterator loc, const TYPE& val);

  beszúrja a val értéket a loc pozíció elé(!), és a beszúrt elemre visszaad egy iterátort

- void **insert**(iterator loc, size_type num, const TYPE& val);

  beszúrja a val-t num-szor a loc pozíció elé

- void **insert**(iterator loc, input_iterator start, input_iterator end);

  a loc pozíció elé beszúrja a start és end között talált elemeket

- void **push_back**(const TYPE& val);

  a vector végére szúrja be a val-t

Törlések:

- void **clear**();

  minden elemet töröl

- void **pop_back**();

  utolsó elemet törli

- iterator **erase**(iterator loc); iterator **erase**(iterator start, iterator end);

  kitörli a loc pozición, vagy a start és end között lévő elemeket, és visszaadja a legutolsó törölt elem utáni pozíciót

Iterátorok:

- iterator **begin**(); const_iterator **begin**() const;

  iterátorok az első elemre

- iterator **end**(); const_iterator **end**() const;

  iterátorok az utolsó utáni(!) elemre

- reverse_iterator **rbegin**(); const_reverse_iterator **rbegin**() const;

  visszaadja hátulról az első elempozíciót (vagyis az utolsót) olyan iterátorként, ami fordított bejárást tesz lehetővé (const és nem const verzió)

- reverse_iterator **rend**(); const_reverse_iterator **rend**() const;

  visszaadja hátulról az utolsó utáni elempozíciót (vagyis az első előttit) olyan iterátorként, ami fordított bejárást tesz lehetővé (const és nem const verzió)

Zh-k, amikben használtuk: hashtable

1.1.2. Deque

A két végű sor (double-ended-queue, röviden: deque) nagyban hasonlít a vector-ra. Szintén tetszőleges eleméhez hozzáférést biztosít, továbbá konstans idejű beszúrást tesz lehetővé mind a konténer elején, mind pedig a végén. A tároló közepén továbbra is lineáris idejűek ezek a műveletek.
Használat: <**queue**> fejállományban
template < **class T, class Allocator** = **allocator**<**T**> > **class deque;**
ahol T a tárolt elemek típusa, Allocator pedig azok foglalásának módja

Tagfüggvények:

Konstruktorok:

1. Ugyanazok, mint a vector-nál, csak persze itt deque szerepel a vector helyett

Operátorok:

1. Ugyanazok, mint a vector-nál

   Leírás:

   két deque akkor számít egyenlőnek, ha elemszámuk és elemeik páronként (==) megegyeznek.

Egyéb tagok: lásd mint a vector-nál +

- void **push_front**(const T&)

  elem beszúrása a tároló elejére

- void **pop_front**()

  elem kivétele a tároló elejéről

Zh-k, amikben használtuk: StackQue

1.1.3. List

Fejelem nélküli, aciklikus, kétirányú láncolt lista. Konstans idejű beszúrást és elemeltávolítást tesz lehetővé mind az elején végén, vagy épp a közepén.
Használat: <**list**> fejállományban
template < **class T, class Allocator** = **allocator**<**T**> > **class list;**
ahol T a tárolt elemek típusa, Allocator pedig azok foglalásának módja

Tagfüggvények:

Konstruktorok:

1. **list**();
2. **list**(const list& c);
3. **list**(size_type num, const TYPE& val = TYPE());
4. **list**(input_iterator start, input_iterator end);
5. **~list**();

   Leírás:

   1. A default ctor: vigyázzunk arra, hogy ekkor a lista még üres
   2. A copy ctor
   3. Megadhatunk egy darabszámot, mennyi elemet szúrjon be létrehozásakor a lista, és egy adott tárolt értéket (egyébként a tárolt típus default ctor()-val hozzuk létre az elemeket)
   4. Két iterátort váró konstruktor: a két iterátor által kijelölt intervallum elemeivel hozzuk létre az új listát
   5. Destruktor

Operátorok:

1. list **operator**=(const list& c2);
2. bool **operator**==(const list& c1, const list& c2);
3. bool **operator!**=(const list& c1, const list& c2);
4. bool **operator**<(const list& c1, const list& c2);
5. bool **operator**>(const list& c1, const list& c2);
6. bool **operator**<=(const list& c1, const list& c2);
7. bool **operator**>=(const list& c1, const list& c2);

   Leírás:

   Ami innen különösen fontos az 1. értékadás operátor (hogy helyesen meg van írva)

   Többihez: két lista akkor számít egyenlőnek, ha elemszámuk és elemeik páronként (==) megegyeznek.

Egyéb tagok:

- bool **empty**() const;

  visszaadja üres-e a lista
- size_type **size**() const;

  tárolt elemek száma

Elem lekérdezések:

- TYPE& **front**(); const TYPE& front() const;

  legelső elem lekérdezése const és nem const módon (az első felülírást is lehetővé tesz pl.: l.front()=7 adott esetben)

- TYPE& **back**(); const TYPE& back() const;

  legutolsó elem lekérdezése const és nem const módon (az első felülírást is lehetővé tesz pl.: l.back()=4 adott esetben)

Beszúrások:

- void **assign**(size_type num, const TYPE& val);

  hozzáad num darab val értékű tárolt elemet a listához

- void **assign**(input_iterator start, input_iterator end);

  az iterátorok által kijelölt tartomány elemeit adja hozzá

- iterator **insert**(iterator loc, const TYPE& val);

  beszúrja a val értéket a loc pozíció elé(!), és a beszúrt elemre visszaad egy iterátort

- void **insert**(iterator loc, size_type num, const TYPE& val);

  beszúrja a val-t num-szor a loc pozíció elé

- void **insert**(iterator loc, input_iterator start, input_iterator end);

  a loc pozíció elé beszúrja a start és end között talált elemeket ([start,end))

- void **push_front**(const TYPE& val);

  a lista elejére szúrja be val-t

- void **push_back**(const TYPE& val);

  a lista végére szúrja be a val-t

Törlések:

- void **clear**();

  minden elemet töröl

- void **pop_front**()

  a legelső elemet törli

- void **pop_back**();

  utolsó elemet törli

- iterator **erase**(iterator loc); iterator **erase**(iterator start, iterator end);

  kitörli a loc pozíción, vagy a start és end között lévő elemeket, és visszaadja a legutolsó törölt elem utáni pozíciót

- void **remove**(const T& val);

  val összes előfordulását törli a listából

- void **unique**();

  minden értékből csak egy előfordulást hagy meg

Iterátorok:

- iterator **begin**(); const_iterator **begin**() const;

  iterátorok az első elemre

- iterator **end**(); const_iterator **end**() const;

  iterátorok az utolsó utáni(!) elemre

- reverse_iterator **rbegin**(); const_reverse_iterator **rbegin**() const;

  visszaadja hátulról az első elempozíciót (vagyis az utolsót) olyan iterátorként, ami fordított bejárást tesz lehetővé (const és nem const verzió)

- reverse_iterator **rend**(); const_reverse_iterator **rend**() const;

  visszaadja hátulról az utolsó utáni elempozíciót (vagyis az első előttit) olyan iterátorként, ami fordított bejárást tesz lehetővé (const és nem const verzió)

Egyéb:

- void **reverse**();

  megfordítja a listában található elemek sorrendjét

- void **sort**();

  rendezi a listában található elemeket

- template<class BinaryPredicate> void **sort**(BinaryPredicate comp);

  hasonló, csak itt átadható a rendezés módja

- void **merge**(list<T, Alloc>& x);

  összefésülés az x listával

- template<class Predicate> void **remove_if**(Predicate p);

  az összes olyan listaelem eltávolítása, amelyre igaz p

Zh-k, amikben használtuk: hashtable

1.2. Asszociatív tárolók

Az asszociatív konténerek változó méretű konténerek, amelyek elemek (értékek) hatékony visszakeresését támogatják kulcsok alapján. Támogatott az elemek beszúrása, törlése, de a szekvenciáktól különböznek abban, hogy az elemeket nem meghatározott helyekre, pozíciókra szúrhatjuk be.

1.2.1. Set

Az STL halmazt megvalósító konténere. A tárolt kulcs típusú objektumokon adva kell legyen valamiféle rendezésnek.
Használat: <**set**> fejállományban
template < **class Key, class Compare** = **less**<**Key**>**, class Allocator** = **allocator**<**Key**> >
class set;
ahol Key típusú elemeket tárolunk, Compare az elemek közti sorrendet definiálja, Allocator pedig az elemek foglalásának módját. (utóbbi kettő default paraméterrel rendelkezik)

Tagfüggvények:

Konstruktorok:

1. **set**();
2. **set**(const set& c);
3. **set**(const key_compare& comp)
4. **set**(input_iterator start, input_iterator end);
5. **~set**();

   Leírás:

   1. A default ctor
   2. A copy ctor
   3. Üres halmazt hoz létre, melyen az elemeket a comp-pal hasonlítjuk össze
   4. Két iterátort váró konstruktor: a két iterátor által kijelölt intervallum elemeivel hozzuk létre az új halmazt
   5. Destruktor

Operátorok:

1. set **operator**=(const set& c2);
2. bool **operator**==(const set& c1, const set& c2);
3. bool **operator!**=(const set& c1, const set& c2);
4. bool **operator**<(const set& c1, const set& c2);
5. bool **operator**>(const set& c1, const set& c2);
6. bool **operator**<=(const set& c1, const set& c2);
7. bool **operator**>=(const set& c1, const set& c2);

   Leírás:

   Ami innen különösen fontos az 1. értékadás operátor (hogy helyesen meg van írva)

   Többihez: két halmaz akkor számít egyenlőnek, ha elemszámuk és elemeik páronként (==) megegyeznek.

Egyéb tagok:

- bool **empty**() const;

  visszaadja üres-e a halmaz
- size_type **size**() const;

  tárolt elemek száma

Elem lekérdezések:

- size_type **count**(const key_type& key) const;

  visszaadja egy elemről, hogy hányszor van benne a halmazban (multiplicitás)

- iterator **find**(const key_type& k) const

  megkeresi a k kulcsú elemet, és visszaad egy iterator-t (ha nincs a halmazban az elem, akkor az end()-et adja vissza)

- iterator **lower_bound**(const key_type& k) const

  megkeresi a legelső elemet, melynek kulcsa **nem kisebb**, mint k

- iterator **upper_bound**(const key_type& k) const

  megkeresi a legelső elemet, melynek kulcsa **nagyobb**, mint k

- pair<iterator, iterator> **equal_range**(const key_type& key) const;

  visszaadja egy elem legelső előfordulásának pozícióját és utolsó előfordulása utáni pozíciót egy pair-ben (pl. 2 4 5 5 5 7 tárolt elemek esetén az első iterátor a 3. elemre mutatna, míg a második iterátor a 6.-ra equal_range(5) hívás esetén)

Beszúrások:

- pair<iterator, bool> **insert**(const value_type& x);

  x-et beszúrja a halmazba, és visszaadja, hogy sikerült-e, illetve a helyét

- iterator **insert**(iterator pos, const value_type& x);

  pos elé beszúrja x-et, és vissza adja az új elem helyét

- template <class InputIterator> void **insert**(InputIterator start, InputIterator end)

  az iterátorok közti elemeket beszúrja a halmazba

Törlések:

- void **clear**();

  minden elemet töröl

- iterator **erase**(iterator loc); iterator **erase**(iterator start, iterator end);

  kitörli a loc pozición, vagy a start és end között lévő elemeket, és visszaadja a legutolsó törölt elem utáni pozíciót

- size_type **erase**(const key_type& key);

  adott kulcsú elem minden előfordulását eltávolítja a halmazból

Iterátorok:

- iterator **begin**(); const_iterator **begin**() const;

  iterátorok az első elemre

- iterator **end**(); const_iterator **end**() const;

  iterátorok az utolsó utáni(!) elemre

- reverse_iterator **rbegin**(); const_reverse_iterator **rbegin**() const;

  visszaadja hátulról az első elempozíciót (vagyis az utolsót) olyan iterátorként, ami fordított bejárást tesz lehetővé (const és nem const verzió)

- reverse_iterator **rend**(); const_reverse_iterator **rend**() const;

  visszaadja hátulról az utolsó utáni elempozíciót (vagyis az első előttit) olyan iterátorként, ami fordított bejárást tesz lehetővé (const és nem const verzió)

1.2.2. Map

Az STL asszociatív tömbjeként, szótáraként szokták emlegetni. Kulcs-érték párosokat tartalmaz, és legfeljebb egy bejegyzés létezhet ugyanazzal a kulccsal. A kulcsok típusán kell legyen egy rendezés.
Használat: <**map**> fejállományban
template < **class Key, class T, class Compare** = **less**<**Key**>, **class Allocator** = **allocator**<**pair**<**const Key,T**> > >
class map;
ahol Key a kulcs típus, T az értékek típusa, Compare a rendezést definiálja, Allocator pedig az elemek foglalásának módját

Tagfüggvények:

Konstruktorok:

1. **map**();
2. **map**(const map& m);
3. **map**(iterator start, iterator end);
4. **map**(iterator start, iterator end, const key_compare& cmp);
5. **map**(const key_compare& cmp);
6. **~map**();

   Leírás:

   1. A default ctor
   2. A copy ctor
   3. Két iterátort váró konstruktor: a két iterátor által kijelölt intervallum elemeivel hozzuk létre az új map-et
   4. Uaz, mint 3. + még a kulcsok közti rendezést is megadhatjuk
   5. Üres map-et hoz létre, melyen az elemek kulcsait a comp-pal hasonlítjuk össze
   6. Destruktor

Operátorok:

1. TYPE& **operator[]**(const key_type& key);
2. map **operator**=(const map& c2);
3. bool **operator**==(const map& c1, const map& c2);

4. bool **operator!**=(const map& c1, const map& c2);
5. bool **operator**<(const map& c1, const map& c2);
6. bool **operator**>(const map& c1, const map& c2);
7. bool **operator**<=(const map& c1, const map& c2);
8. bool **operator**>=(const map& c1, const map& c2);

   Leírás:

   Ami innen különösen fontos az az 1. operátor: ha bent van a tárolóban a key kulcsú elem, akkor az érték mezőt kapjuk meg referencia szerint, de ha még nincs, akkor készül egy ilyen kulcsú bejegyzés, és szintén kapunk egy referenciát az érték mezőre, ahová egyébként a tárolt T típus default ctor()-ával létrehozott elem került

   Illetve az a 2. értékadás operátor (hogy helyesen meg van írva).

   Többihez: két map akkor számít egyenlőnek, ha elemszámuk és elemeik páronként (==) megegyeznek.

Egyéb tagok:

- bool **empty**() const;

  visszaadja üres-e a map

- size_type **size**() const;

  tárolt elemek száma

Elem lekérdezések:

- size_type **count**(const key_type& key) const;

  visszaadja egy kulcsról, hogy hány olyan kulcsú bejegyzés van a map-ben (0 v. 1)

- iterator **find**(const key_type& k)

  megkeresi a k kulcsú elemet, és visszaad egy iterator-t (ha nincs a halmazban az elem, akkor az end()-et adja vissza)

- const_iterator **find**(const key_type& k) const

  mint az előbbi, csak const verzió

- iterator **lower_bound**(const key_type& k) (const verzióban is van)

  megkeresi a legelső elemet, melynek kulcsa **nem kisebb**, mint k

- iterator **upper_bound**(const key_type& k) (const verzióban is van)

  megkeresi a legelső elemet, melynek kulcsa **nagyobb**, mint k

- pair<iterator, iterator> **equal_range**(const key_type& key) (const verzióban is van)

  visszaadja egy key kulcsú elem legelső előfordulásának pozícióját és utolsó előfordulása utáni pozíciót egy pair-ben

Beszúrások:

- pair<iterator, bool> **insert**(const value_type& x)

- iterator **insert**(iterator pos, const value_type& x)
- template <class InputIterator> void **insert**(InputIterator, InputIterator)

  hasonlóak, mint a set-nél, de figyeljünk arra, hogy itt a value_type pair<Key,T> (!!)

Törlések:

- void **clear**();

  minden elemet töröl
- iterator **erase**(iterator loc); iterator **erase**(iterator start, iterator end);

  kitörli a loc pozíción, vagy a start és end között lévő elemeket, és visszaadja a legutolsó törölt elem utáni pozíciót
- size_type **erase**(const key_type& key);

  adott kulcsú elemet eltávolítja a map-ből

Iterátorok:

- iterator **begin**(); const_iterator **begin**() const;

  iterátorok az első elemre
- iterator **end**(); const_iterator **end**() const;

  iterátorok az utolsó utáni(!) elemre
- reverse_iterator **rbegin**(); const_reverse_iterator **rbegin**() const;

  visszaadja hátulról az első elempozíciót (vagyis az utolsót) olyan iterátorként, ami fordított bejárást tesz lehetővé (const és nem const verzió)
- reverse_iterator **rend**(); const_reverse_iterator **rend**() const;

  visszaadja hátulról az utolsó utáni elempozíciót (vagyis az első előttit) olyan iterátorként, ami fordított bejárást tesz lehetővé (const és nem const verzió)

Zh-k, amiben előfordult: bag, ammatrix2

1.3. Adapterek

1.3.1. Stack

Az STL verem adatszerkezetet megvalósító konténere. A Stack csak a legfelső elem módosítását (felülírás, beszúrás, törlés) engedélyezi, egyszóval egy LIFO (last in first out) adatszerkezet. A Stack elemein nem lehetséges végigiterálni.
Használat: <**stack**> fejállományban
template < class T, class Container = deque<T> >
class stack;
ahol T a tárolt elemek típusa, Container pedig az elemek tárolására szolgáló konténertípus

Tagfüggvények:

Konstruktorok:

1. **stack**();

2. **stack**(const stack&);
3. **~stack()**;

   Leírás:

   1. A default ctor
   2. A copy ctor
   3. Destruktor

Operátorok:

1. stack& **operator**=(const stack&);
2. bool **operator**==(const stack&, const stack&)
3. bool **operator**<(const stack&, const stack&)

   Leírás:

   Természetesen ezen a típuson is meg van írva az értékadás operátor.

   Megj.: két stack akkor egyenlő (==), ha elemszámuk megegyezik, és elemeik páronként egyenlőek

Egyéb tagok:

- bool **empty**() const;

  visszaadja üres-e a verem
- size_type **size**() const;

  tárolt elemek száma

Elem lekérdezések:

- value_type& **top**(); const value_type& **top**() const

  felső elem referenciáját adja vissza (const és nem const változat)

Beszúrások:

- void **push**(const value_type&);

  legfelülre szúr be egy elemet

Törlések:

- void **pop**();

  törli a legfelső elemet

1.3.2. Queue

Az STL sor adatszerkezetet megvalósító konténere. A Queue csak a legutolsó pozícióra való beszúrást, és a legelső pozícióról való törlést engedélyez, valamint az első és az utolsó elem lekérdezését, módosítását, egyszóval ez egy "first in first out" (FIFO) adatszerkezet. A Queue elemein nem lehetséges végigiterálni.
Használat: <**queue**> fejállományban
template < **class T, class Container** = **deque**<**T**> > **class queue;**
ahol T a tárolt elemek típusa, Container pedig az elemek tárolására szolgáló konténertípus

Tagfüggvények:

Konstruktorok:

1. **queue**();
2. **queue**(const queue&);
3. **~queue**();

   Leírás:

   1. A default ctor
   2. A copy ctor
   3. Destruktor

Operátorok:

1. queue& **operator**=(const queue&)
2. bool **operator**==(const queue&, const queue&)
3. bool **operator**<(const queue&, const queue&)

   Leírás:

   Természetesen ezen a típuson is meg van írva az értékadás operátor.

   Megj.: két queue akkor egyenlő (==), ha elemszámuk megegyezik, és elemeik páronként egyenlőek

Egyéb tagok:

- bool **empty**() const;
  visszaadja üres-e a sor
- size_type **size**() const;
  tárolt elemek száma

Elem lekérdezések:

- value_type& **front**(); const value_type& **front**() const
  első elem referenciáját adja vissza (const és nem const változat)
- value_type& **back**(); const value_type& **back**() const
  utolsó elem referenciáját adja vissza (const és nem const változat)

Beszúrások:

- void **push**(const value_type&);
  legvégére szúr be egy elemet

Törlések:

- void **pop**();
  törli a legelső elemet

2. Algoritmusok

3. Iterátorok (bejárók)