STL konténerek rövid összefoglalo
Amennyiben pontatlanságot hibát találsz kérlek jelezd!

2016

1. Konténerek

Egy konténer általánosan egy tároló ami objektumok egy gyűjteményét(ezek az elemek) tárolja.

A konténerek szerepe, hogy kezeljék az elemek tároását és tagfüggvényeken keresztül hozzáférést nyújtanak az elemekhez(direkt módon, vagy iterátorokon keresztül).

Általánossgban ezek a konténerek nagyon gyakran alkalmazhatóak feladatok megoldásakor (ezért más nyelvekben is megtalálható a megfelelőjük).

A leggyakoribb konténerek: dinamikus tömbök (*vector*), sorok (*queue*), vermek (*stack*), kupacok (*priority_queue*), láncolt listák (*list*), fa struktúrák (*set*), asszociatív tömbök (*map*).

A feladathoz a megfelelő konténer választása jelentps előnyökkel járhat(pl lista vagy vektor használata olyan esetben mikor sokat akarunk keresni az elemek között többszörös futási időt eredményezhet asszociatív tömbök,vagy set alkalmazásához képest).Itt nem csak a nyújtott funkcionalitást kell figyelembe vennünk,hanem azt, hogy az egyes konténerek ezeket a funkciókat mennyi idő(lépés) alatt tudják teljesíteni(komplexitás).

Ezek közül a félév során dinamikus tömbökkel, listákkal, set-tel és asszociatív tömbökkel foglalkozunk.

2. :: *iterator*

Az STL konténerek esetén az elemekhez való hozzáférés iterátorokon keresztül történik. Az iterátorok referencia típusok pointerekhez hasonló működéssel.

Általában konténerektől két iterátort tudunk lekérni:

- Az első elemhez tartozó iterátort (.*begin*())
- Az utolsó elem utáni iterátort (.*end*())

```
vector<T>::iterator it = myVector.begin();
list<T>::iterator it = myList.begin();
set<T>::iterator it = mySet.begin();
map<K,V>::iterator it = myMap.begin();

vector<T>::iterator it = myVector.end();
list<T>::iterator it = myList.end();
set<T>::iterator it = mySet.end();
map<K,V>::iterator it = myMap.end();
```

Az iterátorokkal végezhető műveletek:

- dereferálás ($*$)
  $*it$ jelentése, az iterátorhoz tartozó elemre való hivatkozás(hasonlóan, mint pointerek esetében). Fontos, hogy ha a dereferált objektum tagfüggvényére, vagy mezőjére szeretnénk hivatkozni, akkor zárójelet kell használni: $(*it).tagfv()$;. Ennek az írásmódnak a rövidítésére szolgál a $->$ jelölés: $it->tagfv()$;.
- inkrementálás ($++$)
  Az iterátor inkrementálása a konténerkövetkező elemére állítja a referenciát. $it++$ vagy $++it$.

- dekrementálás ($--$)
  Az inkrementáláshoz hasonló működés, csak az eggyel korábbi elemre állítja a referenciát.

- *advance*
  Az iterátor értékének növelése több lépéssel. Figyelnünk kell, hogy az iterátor értéke a növelés után is érvényes legyen!! $advance(it, n)$.

- egyenlőség vizsgálat ($==, !=$)
  Két iterátor egyenlőségét vizsgálhatjuk vele. Fontos, hogy nem tudunk egymáshoz viszonított helyzetet ellenőrizni iterátorok között, csak egyenlőséget. Emiatt mint majd látjuk általában egy konténeren a következő képpen iterálhatunk végig:

```
for(kontener<T>::iterator it = myKontener.begin();
                  it != myKontener.end(); ++it){
        //adott elemek feldolgozasa.
}
```

3. *vector* $< T >$

Szekvenciális adatszerkezet, azaz az elemek között egy sorrend van definiálva. A *vector* esetében az elemek amemóriában közvetlenül egymás mellett helyezkednek el. Azaz egy tömbben tároljuk az adatainkat. A *vector* esetében lehetőségünk van elemeket dinamikusan hozzáadni, vagy elvenni a tömbből. Ezek alapján adódik a probléma: lefoglalunk egy adott méretű tömböt a memóriában, de mi van ha több elemt szeretnénk beszúrni mint amennyi ez az adott méret? Erre a megoldás a következő: ha megtellik a lefoglalt tömbünk kényetelenek vagyunk újra lefoglalni egy nagyobbat, majd oda átmásolni az eddigi elemeket (annak érdekében, hogy az alap struktúrátmegtartsuk azaz egymás mellett egy tömbben tárolhassuk az elemeket). Ez rengeteg időt vesz igénybe, pontosabban n elemű vector esetében $O(n)$ lépés szükséges. Azonban mivel tudjuk az adott elemek pozícióját ezért konstans időben tudunk hozzáférni az elemekhez($O(1)$). Hasonlóan a szerkezetből következik, hogy általában, ha nincs rendezve a tömbünk akkor adott elem megkeresése is $O(n)$ komplexitású.
Elemek törlése esetén a tömbbeli folytonosság fenntartásához a törlendő elem utáni elemeket egy pozícióval előrébb kell másolnunk. Azaz a törlés költsége a törlendő elem utáni elemek számától függ. Így a törlés komplexitása is $O(n)$(ha nem fontos az elemek sorrendje akkor lehet javítani ezen). Hasonlóan működik az elem beszúrása a *vector* közepébe.

Tehát *vector*-t akkor érdemes használni, ha gyakran szeretnénk index alapján hozzáférni az elemekhez, de lehetőleg ritkán változtatjuk meg az elemek számát (hiszen mind a törlés mind a beszúrás $O(n)$ idejű).

3.1. Használat

```
#include <vector>
//konstruktorok:
vector<int> v;
```

```
//hossz megadasa:
vector<int> v(7);
//hossz es kezdeti ertek:
vector<int> v(7,2);
//bonyolultabb pelda 7*4-es matrix csupa kettessel
vector<vector<int>> v(7,vector<int>(4,2));
//c++11 ota lehetoseg elemek felsorolasaval:
vector<int> v = {5, 4, 2, 3, 1};
//elem hozzafuzes a vector vegehez
v.push_back(5);
//elem hozzaferes index alapjan:
v[3];
//utolso elem torlese:
v.pop_back();
//meret lekerdezese:
size_t size= v.size();
//annak lekerdezese, hogy urese a vector:
bool urese = v.empty();
//vector meretenek beallitasa
int meret = 10;
v.resize(meret);
//vector elso es utolso elemehez hozzaferes:
int elso = v.front()
int utolso = v.back()
//vector teljes tartalmanak torlese:
v.clear();
//elem hozzaferes iteratorokkal:
vector<int>::iterator it = v.begin();
vector<int>::iterator vege = v.end();
//vegigiteralas:
for(size_t i = 0; i < v.size(); i++){
        //v[i] feldolgozasa
}
//iteratorral:
for(vector<int>::iterator it = v.begin();
                it != v.end(); it++){
        // (*it) feldolgozasa
        cout << *it;
}
//c++11 for each ciklus:
for(int &a : v){
        //'a' lesz v eleme
        cout << a;
}
//elem beszurasa az 5. helyre:
vector<int>::iterator it = v.begin();
advance(it,5);
v.insert(it,4);
//n elem beszurasa:
v.insert(it, n, 3);
```

```
//elem torlese:
v.erase(it);
//tartomany torlese:
v.erase(elso_iterator, utolso_iterator);
```

4. *list* < *T* >

Szintén szekvenciális adatszerkezet, de itt az elemek memóriában nem egymás mellett helyezkednek el. Minden elem esetén tudjuk, hogy ki a megelőző és ki a következő elem. Azaz, ha az i-edik elemet szeretnénk megkeresni, akkor először az elsőt nézzük onnan tudjuk a másodikat, majd a másodiktól harmadikat és így tovább i lépés szükséges az eléréshez. Azonban a láncolást könnyű megváltoztatni, azaz ha az elejére szeretnénk elemet hozzáadni, akkor csak az első elemnek kell megmondani, hogy van őt megelőző elem (és a lista fejét beállítani az első elemre)ez $O(1)$ időben történik, míg a *vector* esetében ez $O(n)$ volt.
Ezekből következően az index alapú elemhozzáférés láncolt lista esetében $O(n)$ idejű. Az elem beszúrás, törlés, hozzáfűzés minden esetben(a pozíció meghatározása után)a láncolások beállításával történik azaz $O(1)$.

4.1. Használat

```
#include <list>
//konstruktorok:
list<int> first;   // empty list of ints
// 4 int 100 ertekkel
list<int> second (4,100);
// second alapjan vegigiteralva
list<int> third (second.begin(),second.end());
// third masolata
list<int> fourth (third);
//c++11 ota lehetoseg elemek felsorolasaval:
list<int> l = {5, 4, 2, 3, 1};
//elem hozzafuzes a vegehez
l.push_back(5);
//elem hozzafuzes az elejehez
l.push_front(4);
//utolso elem torlese:
l.pop_back();
//elso elem torlese:
l.pop_front();
//meret lekerdezese:
size_t size= l.size();
//annak lekerdezese, hogy urese:
bool urese = l.empty();
//elso es utolso elemehez hozzaferes:
int elso = l.front()
int utolso = l.back()
```

```
//teljes tartalmanak torlese:
l.clear();
//elem hozzaferes iteratorokkal:
list<int>::iterator it = l.begin();
list<int>::iterator vege = l.end();
//iteratorral:
for(list<int>::iterator it = l.begin();
                  it != l.end(); it++){
        // (*it) feldolgozasa
        cout << *it;
}
//c++11 for each ciklus:
for(int &a : l){
        //'a' lesz v eleme
        cout << a;
}
//elem beszurasa az 5. helyre:
list<int>::iterator it = l.begin();
advance(it,5);
l.insert(it,4);
//n elem beszurasa:
l.insert(it, n, 3);
//elem torlese:
l.erase(it);
//tartomany torlese:
l.erase(elso_iterator, utolso_iterator);
//ertek alapjan elem torlese:
l.remove(5);
```

5. $set < T >$

A set elemei egy halmazt alkotnak. A halmazokban minden elem maximum egyszer szerepelhet és nincs sorrend definiálva az elemek között, így logikailag nincs értelme a halmaz elejének és végének. A c++ esetében a set mögött egy fa struktúra áll. Erről nekünk elegendő annyit tudnunk, hogy az elemhozzáférés, törlés és beszúrás is $O(log(n))$ idő alatt történik. A fa struktúra megfelelő működéséhez szükséges, hogy a tárolni kívánt típushoz létezzen $<$ operátor (ez alapján történik a keresés és az elemek tárolása is).

5.1. Használat

```
#include <set>
//konstruktorok:
set<int> first;   // empty set of ints
int myints[]= {10,20,30,40,50};
set<int> second (myints,myints+5);
// second alapjan vegigiteralva
set<int> third (second.begin(),second.end());
```

```
// third masolata
set<int> fourth (third);
//c++11 ota lehetoseg elemek felsorolasaval:
set<int> s = {5, 4, 2, 3, 1};
//meret lekerdezese:
size_t size= s.size();
//annak lekerdezese, hogy urese:
bool urese = s.empty();
//teljes tartalmanak torlese:
s.clear();
//elem hozzaferes iteratorokkal:
set<int>::iterator it = s.begin();
set<int>::iterator vege = s.end();
//iteratorral:
for(set<int>::iterator it = s.begin();
                  it != s.end(); it++){
        // (*it) feldolgozasa
        cout << *it;
}
//c++11 for each ciklus:
for(int &a : s){
        //'a' lesz v eleme
        cout << a;
}
//elem beszurasa:
//(ha mar letezik a halmazban az
// adott elem nem hozza letre ujra)
s.insert(4);
//elem torlese:
s.erase(it);
//tartomany torlese:
s.erase(elso_iterator, utolso_iterator);
//elem keresese a halmazban:
set<int>::iterator it = s.find(4);
//ha nem talalhato meg az adott elem
//akkor az end()-el ter vissza:
bool isInSet = s.find(4) == s.end();
//annak ellenorzese, hogy az
//adott elem megtalalhato a halmazban:
int n = s.count(4);
//mivel ennek az ereteke 0 vagy 1
//ezert akar logikai ertekkent
//is hasznalhatjuk kozvetlenul:
bool isInSet = s.count(4);
```

6. $map < K, V >$

A map egy asszociatív tömb. Ennek lényege, hogy a megadott kulcsokhoz tartozóan tárolunk értékeket. A kulcsoknak egyedieknek kell lenniük (értelem szerűen az értékek ismétlődhetnek). A *set*-hez hasonlóan itt is szükséges megírni a $<$ operátort ha saját típust szeretnénk kulcsként használni. A c++ esetében a map mögött is egy fa van (piros-fekete fa) ebből következően itt is a beszúrás, elemelérés, törlés $O(log(n))$ alatt történhet.

6.1. Használat

```
#include <map>
//konstruktorok:
map<int,string> myMap;
myMap[0] = "nullas";
myMap[1] = "egyes";
myMap[6] = "kettes";
// second alapjan vegigiteralva
set<int> third (myMap.begin(),myMAp.end());
// third masolata
set<int> fourth (third);
//c++11 ota lehetoseg elemek felsorolasaval:
map<int,string> m1{{1,"012"},{2,"01"},{6,"0121"}};
map<int,string> m2={{1,"012"},{2,"01"},{6,"0121"}};
//meret lekerdezese:
size_t size= m.size();
//annak lekerdezese, hogy urese:
bool urese = m.empty();
//teljes tartalmanak torlese:
m.clear();
//elem hozzaferes iteratorokkal:
map<int,string>::iterator it = m.begin();
map<int,string>::iterator vege = m.end();
//az iteratorok itt kulcs-ertek parokat jelolnek:
int kulcs = it->first;
string ertek = it->second;
//elem hozzaferes kulcs alapjan:
m[kulcs]; //letrehozza ha nincs ilyen kulcs
m.at(kulcs);//hibat dob ha nincs ilyen kulcs

//iteratorral:
for(map<int,string>::iterator it = m.begin();
                it != m.end(); it++){
        // (*it) feldolgozasa
        cout << "kulcs:␣" << it->first
                 << "ertek:␣" << it->second;
}
//c++11 for each ciklus:
for(int &a : m){
```

```
        //'a' lesz m eleme azaz egy pair<K,V>
        cout << "kulcs: " << a.first
                << "ertek: " << a.second;
}
//elem beszurasa:
if(m.count(kulcs)){
        m[kulcs] = ertek;
}
//elem torlese:
m.erase(it);
//tartomany torlese:
m.erase(elso_iterator, utolso_iterator);
//elem keresese a halmazban:
set<int>::iterator it = m.find(4);
//ha nem talalhato meg az adott elem
//akkor az end()-el ter vissza:
bool isInMap = m.find(4) == m.end();
//annak ellenorzese, hogy az
//adott elem megtalalhato a halmazban:
int n = m.count(4);
//mivel ennek az ereteke 0 vagy 1
//ezert akar logikai ertekkent
//is hasznalhatjuk kozvetlenul:
bool isInMap = m.count(4);
```