Hatékony memóriakezelésű, kettősen láncolt lista

A jól ismert adattípus újabb megvalósításával értékes bájtokat takaríthatunk meg.

A gyártók fontos szempontja a kisméretű eszközök előállítási költségének csökkentése, amelynek kapcsán sokszor a memória kisebbre vétele is felmerül. Az egyik lehetőség a mindennapok során széles körben használt absztrakt adattípusok a megszokottól eltérő megvalósításának használata. Ilyen absztrakt adattípus a kettősen láncolt lista is. Írásomban a kettősen láncolt lista egy hagyományos és egy újfajta megvalósítását fogom ismertetni, illetve kitérek a beszúrás, a bejárás és a törlés műveletének végrehajtására is. Szóba kerül az egyes megvalósítások idő- és memóriaigénye is, így össze lehet hasonlítani előnyeiket és hátrányaikat. Az újabb megvalósítás a mutatók távolságára alapul, ezért ez alkalommal mutatótávolság alapú megvalósításnak fogom hívni. Esetében mindegyik csomópontnak csak egy mutatót kell tárolnia, ez a lista előre és hátra irányú bejárhatóságát egyaránt biztosítja. A hagyományos megvalósításnál egy előre irányú mutató mutat a lista következő, valamint egy visszirányú az előző elemére. Az emiatt jelentkező többletterhelés a hagyományos megvalósításnál 66 százalék, míg a mutatótávolság alapúnál 50 százalék. Ha többdimenziós kettősen láncolt listát használunk, például dinamikus rácsozatot akarunk létrehozni, akkor a megtakarítás ennél jóval nagyobb is lehet.
A kettősen láncolt listák hagyományos megvalósítását részletesebben nem fogom elemezni, leírása ugyanis gyakorlatilag bármely adatszerkezetekkel vagy algoritmusokkal foglalkozó könyvben megtalálható. A hagyományos és a mutatótávolság alapú megvalósítás használata nagyon hasonló, így memória- és időigényük is könnyen összehasonlítható.

A csomópont

A mutatótávolság alapú megvalósítás egy csomópontja a következőképpen épül fel:

```
typedef int T;
typedef struct listNode{
        T elm;
        struct listNode * ptrdiff;
};
```

A `ptrdiff` mutató a következő és az előző csomópontra irányuló mutatók közötti távolságot adja meg. A mutatótávolságot kizáró *VAGY (EXOR)* művelettel kapjuk meg. Minden ilyen listának van egy kezdő és egy záró csomópontja.
A kezdő csomópont a lista fejére mutat, a záró pedig a farkára. Definíció szerint a kezdő csomópont előtt egy *NULL* csomópont található, ahogy a záró csomópont után is. Egy egyetlen csomópontból álló listánál az előző és a következő csomópont egyaránt *NULL* csomópont, ezért a `ptrdiff` mezőbe is *NULL* mutató kerül. Egy két csomópontból álló listában a kezdő csomópont előtti csomópont *NULL*, az utána lévő pedig a záró csomópont. A kezdő csomópont esetében a `ptrdiff` mezőbe a záró csomópont és a *NULL* mutatója közötti *EXOR* művelet eredményét kell írnunk, amivel a záró csomópontot kapjuk. A záró csomópont `ptrdiff` mezőjébe hasonló eljárással a kezdő csomópont kerül.

Bejárás

Az egyes csomópontok beillesztésének alapja a bejárás. Az előre- és a hátrafelé mozgáshoz egyetlen egyszerű eljárásra van szükség. Ha átadott értékként a kezdő csomópontot adjuk meg, akkor – mivel az előző csomópont *NULL* – a bejárás értelemszerűen balról jobbra irányul. Ha átadott értékként a záró csomópontot adjuk meg, akkor a bejárás jobbról balra halad. A jelenlegi megvalósítás a lista közepéről indított bejárást nem támogatja, ugyanakkor ennek megvalósítása sem okozhat különösebb nehézséget. A `NextNode` (következő csomópont) a következőképpen épül fel:

```
typedef listNode * plistNode;
plistNode NextNode( plistNode pNode,
                    plistNode pPrevNode){
    return ((plistNode)
      ((int) pNode->ptrdiff ^ ( int)pPrevNode) );
}
```

Minden elem mutatótávolságát úgy számítjuk, hogy *EXOR* műveletet végzünk a következő és az előző csomópont között. Ha tehát *EXOR*-t végzünk az előző csomóponttal, megkapjuk a következőre irányuló mutatót.

Beillesztés

Tegyük fel, van egy új csomópontunk, valamint ismerjük egy meglévő csomópont értéktartalmát, és az új csomópontot a megadott értéket bejárási irány szerint elsőként tartalmazó csomópont után szeretnénk beilleszteni. (*1. kódrészlet*) Meglévő listába új csomópontot három csomópont mutatóinak módosításával illeszthetünk be, ezek a jelenlegi, a jelenlegi után következő és az új csomópont. Ha átadott értékként az utolsó csomópont értékét adjuk meg, akkor a lista végére fogjuk fűz-

1. kódrészlet Új csomópont beillesztésére szolgáló függvény

```
void insertAfter(plistNode pNew, T theElm)
{
   plistNode pPrev, pCurrent, pNext;
   pPrev = NULL;
   pCurrent = pStart;
   while (pCurrent) {
      pNext = NextNode(pCurrent, pPrev);
      if (pCurrent->elm == theElm) {
         /* véget ért a bejárás */
         if (pNext) {
            /* módosítjuk a meglévő következő
            ➥csomópontot */
            pNext->ptrdiff =
                (plistNode) ((int) pNext->ptrdiff
                           ^ (int) pCurrent
                           ^ (int) pNew);
            /* módosítjuk a jelenlegi csomópontot */
            pCurrent->ptrdiff =
              (plistNode) ((int) pNew ^ (int) pNext
                         ^ (int) pCurrent->ptrdiff);
            /* módosítjuk az új csomópontot */
            pNew->ptrdiff =
                (plistNode) ((int) pCurrent
                           ^ (int) pNext);
         break;
      }
      pPrev = pCurrent;
      pCurrent = pNext;
   }
}
```

ni az új csomópontot. Ha ilyen lépések sorozatával építünk fel egy listát, az időigényt is könnyen meg tudjuk vizsgálni. Ha az `InsertAfter()` (beillesztés mögé) eljárás nem találja a megadott értéket, akkor nem illeszt be új elemet.
Először a `NextNode()` eljárás segítségével ellépkedünk a megadott értéket tartalmazó csomópontig. Ha megtaláltuk, akkor mögé illesztjük az új csomópontot. Mivel a következő csomópont mutatókülönbözetet tárol, a megtalált csomóponttal ***EXOR*** kapcsolatba hozva feloldjuk a benne tárolt értéket. Következő lépésként ***EXOR*** műveletet végzünk az új csomópont felhasználásával, hiszen a következő csomópont előző csomópontja az új csomópont lesz. A jelenlegi csomópontot hasonló eljárást követve módosítjuk. Először egy ***EXOR*** művelettel feloldjuk a következő csomópontra vonatkozó mutatókülönbséget. Újabb ***EXOR*** műveletet végzünk az új csomóponttal, ezzel megkapjuk az új mutatókülönbséget. Végül, mivel az új csomópont a jelenlegi és a következő közé kerül, ezeket ***EXOR*** kapcsolatba hozva megkapjuk az új csomópont mutatókülönbségét.

Törlés

A törlés jelenlegi megvalósítása a teljes listát törli. Írásom célja a dinamikus memóriahasználat szemléltetése és a megvalósított primitívek futtatási idejének vizsgálata. Nyilván nem volna nehéz primitív műveletek mindenre kiterjedő gyűjteményével előállni. Mivel a bejáráshoz két csomópontra irányuló mutatót is ismernünk kell, a jelenlegi csomópontot nem törölhetjük azonnal a következő csomópont megtalálása után. Ehelyett, ha megtaláltuk a következő csomópontot, mindig az előzőt töröljük. Ha a jelenlegi csomópont helyének felszabadításakor a jelenlegi csomópont az utolsó, akkor végeztünk is. Egy csomópont akkor számít utolsónak, ha a vele végrehajtott `NextNode()` függvény ***NULL*** csomópontot ad vissza.

Memória- és időigény

Az itt tárgyalt megvalósítás kipróbálására a második kódrészletet a ***Linux Journal*** FTP-helyéről (***ftp.ssc.com/pub/lj/listings/issue129/6828.tgz [1]***) lehet letölteni. Saját ***Pentium II*** (349 MHz, 32 MB RAM és 512 KB másodszintű gyorsítótár) gépemen futtatva a mutatótávolság alapú megvalósítás 15 másodperc alatt hozott létre húszezer csomópontot. Ennyi időre van szükség húszezer csomópont beillesztéséhez.
A teljes lista bejárása vagy törlése egy másodpercig sem tart, ezért ezeket a műveleteket nem ilyen időfelbontásban érdemes vizsgálni. Rendszerszintű megvalósításnál az időzítéseket inkább milliszekundumos felbontással szokták figyelni. Ha ugyanazt a mutatótávolság alapú megvalósítást tízezer csomóponttal futtatjuk, akkor a beillesztés mindössze három másodpercig tart. A teljes lista bejárása vagy törlése kevesebb mint egy másodpercet igényel. Húszezer csomópont tárolásához 160000 bájtra van szükség, tízezer csomópont pedig 80000 bájtnyi területen fér el. 30000 csomópont tárolásakor a beillesztéshez 37 másodpercre van szükség. A teljes lista bejárása vagy törlése ez esetben is egy másodperces időtartam alatt történik meg. Az időigényekből nagyjából látható, hogy a dinamikus memória (kupac) használata egyre erőteljesebb. A dinamikus memóriában egyre nehezebb szabad szakaszt találni, ennek ideje nemlineárisan, sőt, inkább hatványosan növekszik.
A hagyományos megvalósításnál tízezer csomópont beillesztése ugyancsak három másodpercig tart. A bejárás ideje előre és hátra haladva egyaránt egy másodperc alatt van. Tízezer csomópont tárolásához összesen 120000 bájtra van szükség. Húszezer csomópont kezelésekor a beillesztés 13 másodpercet igényel, a bejárás és a törlés ideje pedig továbbra is egy másodpercen belül marad. Húszezer csomópont kezeléséhez összesen 240000 bájtot kell biztosítani. Harmincezer csomópontnál 33 másodpercig tart a beillesztés, és továbbra is kevesebb mint egy másodpercig a bejárás és a törlés. Harmincezer csomópont összesen 360000 bájt területet foglal le.

Összefoglalás

A kettősen láncolt lista hatékony memóriakezelésű változata az időigény tekintetében minimális többlet mellett valósítható meg. Okos tervezéssel mindkét megvalósításhoz össze lehet állítani a primitív műveletek átfogó gyűjteményét, ám a műveletek időigénye jelentős mértékben eltérhet.

Linux Journal 2005. január, 129. szám

Prokash Sinha 18 éve rendszerprogramozással foglalkozik. Dolgozott már fájlrendszereken, hálózat- és memóriakezelésen, UNIX, OS/2, NT, Windows CE és DOS operációs rendszer alatt egyaránt. Fő érdeklődési területe a rendszermag és a beágyazott rendszerek. A prokash@garlic.com címen érhető el.