# A db4o objektumközpontú adatbázismotor beépítése saját alkalmazásainkba

A cikkből azt tudhatjuk meg, miként használhatjuk ezt a kis helyigényű, objektum-orientált adatbázis beágyazott rendszereken futó programokban.

A *db4o* egy nyílt forrású, objektum-orientált adatbázis, amit a *db4object* készít. A rendszer beágyazható abban az értelemben, hogy mivel a teljes *db4o* motor egyetlen könyvtár formájában van megvalósítva, egyszerűen összeszerkeszthetjük a saját programunkkal. A db4o motornak ráadásul van megfelelő változata *Java*-hoz, illetve a *.NET/Mono* rendszerekhez egyaránt. Ennek megfelelően ideális *OODBMS* motor *Linuxon* történő fejlesztéshez, akár *Java*-t, akár a *.NET* keretrendszert használjuk. Az ebben a cikkben bemutatásra kerülő példák *C#* nyelven készültek, és egy *Ubuntu Linux* rendszeren futó *Mono* fordítóval próbáltam ki őket. (Az alkalmazások ezen kívül biztosan futnak még *Monoppix* rendszeren is.)
Amellett, hogy nyílt forrású eszköz lévén egyszerűen letölthetjük és azonnal használni kezdhetjük, a *db4o*-nak megvan az az előnye is, hogy tömör, rendkívül jól átgondolt programozási felületet biztosít a fejlesztőnek. A legtöbb esetben a saját metódusainkat kilenc alapvető hívásból vezethetjük le. Mellesleg a könyvtár memóriaigénye nem különösebben nagy, így kifejezetten jól használható olyan esetekben, amikor a rendelkezésre álló erőforrások korlátozottak. (Ugyanakkor a db4o minden kétséget kizáróan alkalmatlan egész vállalkozások vagy szervezetek kiszolgálására.)
Annak ellenére, hogy a *db4o* egész egyszerűen használható, és az erőforrásigénye is kicsi, gyakorlatilag az összes olyan szolgáltatással rendelkezik, amit egy kereskedelmi adatbázismotortól elvárnánk. Lehetővé teszi a többfelhasználós működést, valamennyi, az adatbázishoz való hozzáférést a felhasználó számára láthatatlan módon tranzakciókba foglal, ezen kívül pedig minden működési folyamata során követi az úgynevezett *ACID* elveket (atoicity, consistency, isolation, durability).
Számos más, objektum-orientált és objektum-relációs adatbázismotortól eltérően a db4o használatakor nem kell az általunk írt kódot elő- vagy utófeldolgozásnak alávetni. Arra sincs szükség, hogy az állandó osztályokat egy speciális, az állandóságot külön kezelő szuperosztályból vezessük le. A *db4o* kiválóan működik a közönséges objektumokkal, és azt sem kell vele előre közölnünk, hogy milyen egy osztály belső szerkezete, mielőtt az ebbe tartozó objektumot letárolnánk azt egy *db4o* adatbázisban.

Amint ez a cikkből is hamarosan kiderül, mindezek a tulajdonságok néhány különleges képességgel is felruházzák ezt a fejlesztőeszközt.

## Egy szótár-adatbázis kialakítása

Tegyük fel, hogy az alkalmazás, amit fejlesztünk egy szótár, mégpedig a szó klasszikus értelmében. A programnak tehát egy olyan adatbázist kell tudnia kezelni, amelyek szavakat, és azok meghatározását tartalmazza. Egy ilyen alkalmazásnál a szótári bejegyzések modellezésére a következő osztályt adhatjuk meg:

```
/*
 * DictEntry
 */
using System;
using System.Collections;
namespace PersistentTrees
{
        /// <summary>
        /// DictEntry class
        /// A dictionary entry
        /// </summary>
        public class DictEntry
        {
                private string theWord;
                private string pronunciation;
                private ArrayList definitions;

                public DictEntry()
                {
                }

                // Create a new Dictionary Entry
                public DictEntry(string _theWord,
                                string
➥_pronunciation)
                {       theWord = _theWord;
                        pronunciation =
➥_pronunciation;
                        definitions = new
➥ArrayList();
```

© Kiskapu Kft. Minden jog fenntartva

© Kiskapu Kft. Minden jog fenntartva

```
                }

                // Add a definition to this entry
                // Note that we do not check for
                // duplicates
                public void add(Defn _definition)
                {
                        definitions.Add
➥(_definition);
                }

                // Retrieve the number of
                // definitions
                public int numberOfDefs()
                {
                        return definitions.Count;
                }

                // Clear the definitions array
                public void clearDefs()
                {
                        definitions.Clear();
                        definitions.TrimToSize();
                }

                // Properties
                public string TheWord
                {
                        get { return theWord; }
                        set { theWord = value; }
                }
                public string Pronunciation
                {
                        get { return
➥pronunciation; }
                        set { pronunciation =
➥value; }
                }

                // Get reference to the
                // definitions
                public ArrayList getDefinitions()
                {
                        return definitions;
                }
        }
 }
```

Amint látható, a `DictEntry` objektumoknak összesen három eleme van: maga a szó, annak a kiejtése, valamint egy lista, ami a meghatározásokat tartalmazza. Mindeközben egy osztály, amely a meghatározásokat hordozó objektumot írja le a a következőképpen nézhet ki:

```
/*
 * Defn
 *
 */
using System;

namespace PersistentTrees
{
        /// <summary>
        /// Description of Class1.
        /// </summary>
        public class Defn
        {
                public static int NOUN = 1;
                public static int PRONOUN = 2;
                public static int VERB = 3;
                public static int ADJECTIVE = 4;
                public static int ADVERB = 5;
                public static int CONJUNCTION = 6;
                public static int PARTICIPLE = 7;
                public static int GERUND = 8;

                private int pos;
                private string definition;

                public Defn(int _pos,
                           string _definition)
                {
                        pos = _pos;
                        definition = _definition;
                }

                // Properties
                public int POS
                {
                        get { return pos; }
                        set { pos = value; }
                }
                public string Definition
                {
                        get { return definition; }
                        set { definition = value; }
                }
        }
}
```

Látható, hogy a `Defn` objektum tartalmaz egy egész típusú tagot, amely a kiejtést jelzi, valamint egy karakterláncot, amely a meghatározás szövegét hordozza. Ez a szerkezet lehetővé teszi, hogy egy a szótárban szereplő szóhoz akár több meghatározást is hozzárendeljünk.
Az ilyen típusú elemeknek egy ***db4o*** adatbázisban való letárolása döbbenetesen egyszerű. Tegyük fel, hogy a „float” szót akarjuk felvenni a szótárba, és egyszerre három meghatározást is szeretnénk hozzárendelni. Ezt a következő kóddal érhetjük el.

```
Defn _float1 = new Defn(VERB, "To stay on top of
➥a liquid.");
Defn _float2 = new Defn(VERB, "To cause to
➥float.");
Defn _float3 = new Defn(NOUN, "Anything that
➥stays on top of water.");
DictEntry _float = new DictEntry("float",
➥"flote");
```

```
_float.add(_float1);
_float.add(_float2);
_float.add(_float3);
```

Ezen a ponton tehát már rendelkezünk egy *_float* nevű, *DictEntry* típusú objektummal, amelynek a meghatározás része három bejegyzést tartalmaz.
A bevitelhez először is megnyitjuk magát az adatbázist. Egy *db4o* adatbázist mindig egy `ObjectContainer` objektum modellez, amit viszont a következő módon nyithatunk meg, vagy hozhatunk létre, ha történetesen még nem létezik:

```
ObjectCointainer db =
➥Db4o.openFile("<filename>");
```

Itt `<filename>` annak a fájlnak a neve és elérési útvonala, amely az `ObjectContainer` objektum állandó tartalmát hordozza. Az `ObjectContainer`-be objektumot behelyezni a `set()` metódussal tudunk. Tároljuk le tehát az imént létrehozott új bejegyzést:

```
db.set(_float);
```

Lehet ugyan, hogy így elsőre hihetetlennek tűnik, de ezzel a dolog el is van intézve. Összesen ennyit kell tudnunk a `set()` metódusról. A fenti egyetlen hívás nem csak letárolja a `_float` nevű, `DictEntry` típusú objektumot az adatbázisban, de beleírja a hozzá tartozó összes `Defn` objektumot is. Amikor meghívjuk a *db4o* `set()` metódusát, a rendszer automatikusan, a felhasználó számára láthatatlanul végigpásztázza az adott objektum összes függőségét, felderíti a gyermekobjektumokat, és azokat is megfelelően kezeli.
Ha tehát van egy akármilyen összetett objektumszerkezetünk, amit tárolni szeretnénk, semmi egyebet nem kell tennünk, csak átadni a `set()` metódusnak a szerkezet gyökérobjektumát. A többit a rendszer magától elintézi, és az egész szerkezetet egyetlen hívással letárolja. A *db4o*-nak nem kell külön „elárulnunk” az általunk kezelt objektumok szerkezetét, mivel képes magától felderíteni azt.
Ha nem beírni, hanem lekérdezni szeretnénk egy objektumot az `ObjectContainer`-ből, akkor azt a *db4o* úgynevezett *QBE (query by example)* mechanizmusával határozhatjuk meg. A *QBE* stílusú lekérdezéseket egy minta- vagy sablonobjektum vezérli. Kicsit konkrétabban egy lekérdezést ebben az esetben úgy kell végrehajtanunk, hogy létrehozunk egy mintaobjektumot, majd feltöltjük a mezőit azokkal az értékekkel, amelyekre szeretnénk rákeresni. Ezt aztán átadjuk a rendszernek, és ezzel képletesen azt mondjuk neki: *„Nézd öregem, itt van ez az objektum. Keresem az összes olyan tárolt objektumot, ami pont így néz ki, mint ez itt.”*
Ha tehát az imént létrehozott bejegyzést (float) akarjuk a szótárból visszakerestetni, akkor a következőképpen kell eljárnunk:

```
// Create template
DictEntry DTemplate = new DictEntry("float", "");
// Execute QBE
ObjectSet results = db.get(DTemplate);
// Iterate through results set
while(results.hasNext())
{
  DictEntry _entry = (DictEntry)results.next();
  ... process the DictEntry object ...
}
```

Látható, hogy először is létrehoztunk egy sablonobjektumot, aztán kitöltöttük a mezőit azokkal a dolgokkal, amelyekre kíváncsiak vagyunk. Azokba a mezőkbe, amelyek tartalmára nem vagyunk kíváncsiak nulla került, vagy üres karakterlánc, attól függően, hogy milyen típusú adatról van szó. (A fenti példában mi egyszerűen csak a „float” szót keressük a szótárban. Ennek megfelelően a sablonobjektum konstruktorában üresen hagytuk a kiejtésre vonatkozó mezőt, hiszen ennek most a lekérdezés szempontjából nincs jelentősége.)
Ha készen van a sablon, a lekérdezést az `ObjectContainer` objektum `get()` metódusának meghívásával hajthatjuk végre. Ennek egyetlen argumentumként magát a sablont kell átadni. A lekérdezés eredménye egy *ObjectSet* objektum lesz, amin végighaladva kiolvashatjuk a konkrét találatokat.

## Indexek létrehozása

Ezen a ponton tehát már képesek vagyunk könnyedén létrehozni egy adatbázist, fel tudjuk tölteni azt tartalommal, és le is tudjuk kérdezni az elemeit a *db4o QBE* mechanizmusával. De mi van akkor, ha más, index alapú lekérdezésekkel szeretnénk kísérletezni? Mivel az adatbázis megőrzi az állandó objektumok közti relációkat, felépíthetünk egy olyan egyedi indexet és navigációs struktúrát, ami a tájékozódást segíti. Ezt az indexet aztán szintén elhelyezhetjük magában az adatbázisban úgy, hogy a megfelelő adatobjektumok is hozzá vannak „drótozva”.
Hogy mindez milyen egyszerűen megvalósítható, azt két, erősen eltérő indexelési szerkezet létrehozásával fogom demonstrálni.
Az első példában egy bináris fát fogunk létrehozni. A fa valamennyi csomópontja egy kulcs-érték párt fog tartalmazni. A kulcs egy a szótárban található szó lesz, míg az adattartalom egy hivatkozás lesz arra a *DictEntry* objektumra, ami a kérdéses szót tárolja az adatbázisban. Ezzel a szerkezettel tehát lehetőségünk lesz arra, hogy az adatbázisból először lekérjük magát a bináris fát, azon végrehajtsuk a keresést, majd a megtalált kulcs-érték pár(ok) alapján lekérjük magát a *DictEntry* objektumot is (már ha van találat).
Mivel a bináris fák felépítését és működését az Olvasó bizonyára jól ismeri, itt és most nem is mennék bele ennek a taglalásába. (Ami azt illeti manapság számos keretrendszer szabványos adatszerkezetként tartalmazza az ilyen struktúrákat. Én ennek ellenére explicit módon hoztam létre egy ilyen fát, mivel azt szerettem volna szemléltetni, hogy mennyire könnyű az adatbázisban való letárolása.)
A bináris fa megvalósítását az *1. Lista* tartalmazza. Meglehetősen kezdetleges szerkezetről van szó, ami kizárólag a beillesztést, a lekérdezést és a keresést támogatja. Nem garantálja a fa kiegyensúlyozottságát sem, hiszen csak demonstrációs célokat szolgál, nem akarunk vele semmi komolyat művelni. A *TreeNode* osztály megvalósítása, amely a bináris fa csomópontjainak szerkezetét írja le, a *2. Listában* látható. (Az *1. Listában* felbukkanó `db.activate()` hívások értelmére hamarosan fény derül.)

© Kiskapu Kft. Minden jog fenntartva

© Kiskapu Kft. Minden jog fenntartva

*1. Lista* A bináris fa megvalósítása

```
/*
 * Binary Tree
 */
using System;
using com.db4o;
namespace PersistentTrees
{
        /// <summary>
        /// Description of BinaryTree.
        /// </summary>
        public class BinaryTree
        {
                // The tree's root
                private TreeNode root;
                public BinaryTree()
                {
                        root = null;
                }
                public static BinaryTree
                ➥nullfactory()
                {
                        return(new
                        ➥BinaryTree());
                }
                // insert
                // Add key to tree with
                // associated object reference
                public void insert(string _key,
                ➥Object _data)
                {
                        // Use recursion for
                        // this
                        root = insert(root,
                        ➥_key, _data);
                }
                // insert
                // This is worker method for
                // inserting key and data
                // Insert _key into subtree t
                // with _data associated
                private TreeNode insert(TreeNode
                ➥t, string _key, Object
_data)
                {
                        // If this subtree is
                        // empty, build a new node
                        if(t == null)
                                t = new TreeNode
                                ➥(_key, _data);
                        else

if(_key.CompareTo(t.Key)<=0)
                                        t.Left =
➥insert(t.Left,_key,
_data);
                        else
                                t.Right =
➥insert(t.Right,_key,
_data);
                        return(t);
                }
                // search
                // Search for a key in the tree.
                // Return the array from the
                // TreeNode if found, null if
                // not
                // db is the ObjectContainer
                // holding the tree.
                public Object[] search(string
                ➥_key,
                                Object
➥Container db)
                {
                        TreeNode t =
➥search(root, _key, db);
                        if(t==null)
➥return(null);       // Not found
                        db.activate(t,4);     //
Activate to get data
                        return(t.getData());
                }
                // search
                // This is the worker method for
                // searching.
                private TreeNode search
                ➥(TreeNode t,
                                string
                                ➥_key,
                               Object
➥Container db)
                {
                        // Empty tree?
                        if(t==null)
➥return(null);

if(_key.CompareTo(t.Key)==0) return(t);

if(_key.CompareTo(t.Key)<0)
                        {
                                db.activate
➥(t.Left,2);
                                return(t =
➥search(t.Left,_key,db));
                        }
                        db.activate(t.Right,2);
                        return(t = search
➥(t.Right,_key,db));
                }
        }
}
```

© Kiskapu Kft. Minden jog fenntartva

*2. Lista* A TreeNode osztály megvalósítása

```
/*
 * TreeNode
 */
using System;
namespace PersistentTrees
{
        /// <summary>
        /// Description of TreeNode.
        /// </summary>
        public class TreeNode
        {
                public TreeNode()
                {
                }
                private TreeNode left;  // Left
➥child
                private TreeNode right; // Right
➥child
                private string key;     // Key
➥for this node
                private Object[] data;  // Data
➥associated with key
                // Create a new TreeNode, loaded
                // with
                //  key and data.
                public TreeNode(string _key,
➥Object _data)
                {
                        left = null;
                        right = null;
                        key = _key;
                        data = new Object[1];
                        data[0] = _data;
                }
                // addData
                // Adds new data item to an
                // existing node.
                // The array is extended.
                public void addData(Object
➥_data)
                {
                        Object[] newdata = new
                        ➥Object[data.Length+1];
                        Array.Copy(data,0,
                        ➥newdata,0,data.Length);
                        newdata[data.Length]=
                        ➥_data;
                        data = newdata;
                }
                // Property access
                public TreeNode Left
                {
                        get { return left; }
                        set { left = value; }
                }
                public TreeNode Right
                {
                        get { return right; }
                        set { right = value; }
                }
                public string Key
                {
                        get { return key; }
                        set { key = value; }
                }
                public Object[] getData()
                {
                        return data;
                }
        }
}
```

A következő példában egy olyan speciális indexelő adatszerkezetet fogunk létrehozni, ami kifejezetten szöveges tartalmaz keresésére van kihegyezve. Működésének lényege, hogy szintekre tagolt csomópontok sorozatából épül fel. Minden szinthez egy-egy karakternyi információ tartozik, a szinteződés maga pedig a betűk egymásutániságának felel meg. A legfelső, vagyis a gyökérelem tehát egy tetszőleges szó első betűje lehet, a második szinten levők bármely szó második helyen levő karakterének felelnek meg, és így tovább. A szerkezet elemei tehát úgy követik egymást, mint gyöngyök a gyöngysoron. Egy keresés során nincs más dolgunk, mint szintenként végigmenni felülről lefelé ezen a szerkezeten, amíg teljesen végig nem betűztük a keresett kifejezést.

Ha elsőre kicsit nehéz lenne elképzelni a dolgot, az *1. ábra* talán segíthet benne.

Ha új szót akarunk beilleszteni egy ilyen szerkezetbe, nincs különösebben nehéz dolgunk. Vesszük a szó első betűjét, és megvizsgáljuk a gyökércsomópontot, hogy van-e egyezés. Ha nincs, akkor létrehozzuk az új karakternek megfelelő elemet, majd az algoritmus ettől kezdve új csomópontokat fog beilleszteni a szerkezetbe. (Minden új csomópontot a beilleszteni kívánt szó soron következő betűjével inicializálunk.) Ha a keresett karakter már létezik a szerkezetben, akkor az algoritmus követi a letárolt mutatót, és az alapján elmegy a következő szintre. Ez után a vizsgálat módszere teljesen azonos a korábbival, tehát egyszerűen csak rekurzív módon folytatni kell a vázolt műveletsort, amíg a szó végére nem érünk. Ha nincs több azonosítandó karakter, akkor az a csomópont, ahol éppen tartózkodunk fogja tárolni a megfelelő adatmutatót.

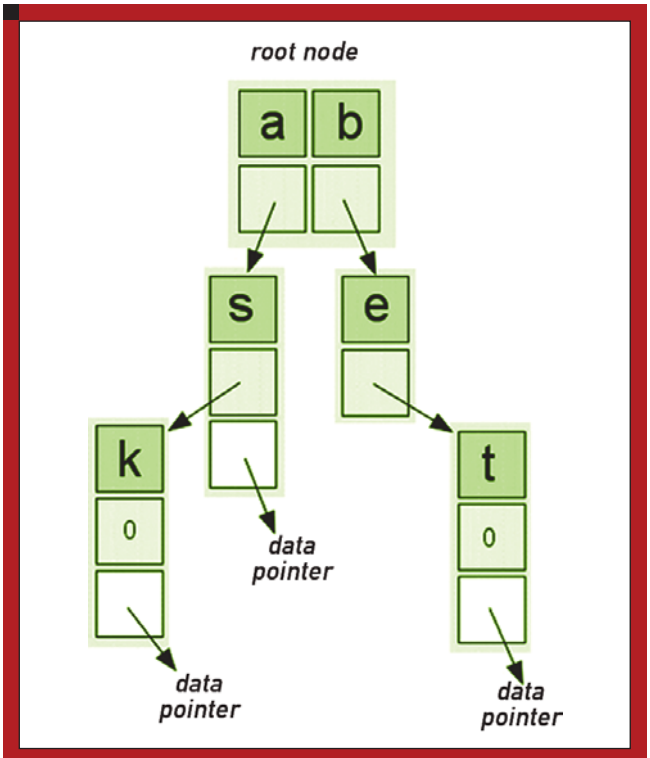
Az ilyen szerkezetű index keresése ugyanilyen egyszerű. Természetesen most is gyökércsomópontnál kezdünk, és megvizsgáljuk, hogy van-e illeszkedés az első karakterre. Ha van, akkor a megfelelő hivatkozás mentén a következő szintre lépünk, és ott folytatjuk a keresést a második betűvel. Ha már az első szinten sincs egyezés, akkor a felhasználó egy „nem található” üzenete kap vissza. Ha a keresés

© Kiskapu Kft. Minden jog fenntartva

során végigértünk a szó összes karakterén, akkor az utolsó csomópontból kiolvassuk a megfelelő `DictEntry` objektum mutatóját, és azt lekérjük az adatbázisból.
A „nyaklánc” megvalósítása a *3. Listában* olvasható.
Amint a fenti két példa is kiválóan mutatja a ***db4o*** segítségével ugyanolyan egyszerűen kezelhetjük az adatbázis-objektumokat, mintha közönséges, a memóriában tárolt objektumok lennének. Ez az állítás pedig független a megvalósítani kívánt adatszerkezettől. Ha pedig indexet szeretnénk létrehozni, nincs más dolgunk, mint a megfelelő adatelemre történő hivatkozást letárolni annak a megfelelő pontján. Ráadásul mivel az adatbázis nem tesz különbséget indexobjektumok és adatobjektumok között, nem kell létrehoznunk külön index- és adatfájlokat. Mindent, amire szükségünk van, egyetlen helyen tárolhatunk, ami sokkal nagyobb előny annál, mint ahogy az ember elsőre képzelné.
A következő kód elolvas egy szavakat és meghatározásokat tartalmazó szöveges fájlt, létrehozza a megfelelő `DictEntry` objektumokat, letárolja azokat az adatbázisban, felépíti a korábban bemutatott bináris fát és „nyaklánc” indexet, és elhelyezi azokban a `DictEntry` objektumok mutatóit:

```
string theword;
string pronunciation;
int numdefs;
int partofspeech;
string definition;
DictEntry _dictEntry;

// Open a streamreader for the text file
FileInfo sourceFile = new FileInfo(textFilePath);
reader = sourceFile.OpenText();
```




**1. ábra** A „nyaklánc” modell. Ez egy olyan index, amelyben a szavak egymást követő karakterei a szerkezet egymást követő szintjeinek felelnek meg. A képen látható index összesen három szót tárol: as, ask, és bet. A csomópontokban szereplő adatmutatók esetünkben azokra a szótárban tárol DictEntry objektumokra mutatnak, amelyek az adott szónak felelnek meg.

*3. Lista* Egy különleges, kifejezetten szöveg alapú keresésre kitalált indexelési szerkezet megvalósítása

```
/*
 * Trie
 */
using System;
using com.db4o;
namespace PersistentTrees
{
        /// <summary>
        /// Description of Trie.
        /// </summary>
        /// trie class
        public class Trie
        {
                private TriePnode root;  // Root
➥of Trie
                // Constructor
                public Trie()
                {
                        root = null;
                }
                // insert
                // Insert a key/data pair into
                // the tree.
                //  Allows duplicates
                public void insert(string
➥key,  // Key to insert
                        Object data)   // Data
➥assoc.
with key
                {
                        TriePnode t = root;
                        TriePnode parent = null;
                        int index=0;
                        int slen = key.Length;
                        for(int i=0; i< slen;
                        ➥i++)
                        {
                                char c = key[i];
                                // If a node
➥doesn't exist – create it
                                if(t == null) t =
➥new TriePnode();
                                // If this is
➥the first node of the tree,
                                // it is the
```

*3. Lista* folytatás

```
                                //  root.
➥Otherwise, it is stored in the
                                //  pnodes array
                                //  of the
➥parent
                                if(i==0)
                                        root =
➥t;
                                else
                                        parent.
➥setPnodePointer(index, t);
                                // If the
➥character is not on the node,
                                // add it
                                if((index=
➥t.isCharOnNode(c))==-1)
                                        index =
➥t.addKeyChar(c);
                                if(i == slen-1)
➥break;
                                parent = t;
                                t =
t.getPnodePointer(index);
                        }
                        // Finally, add the data
                        // item
                        t.addData(index, data);
                }
                // search
                // Searches for a string in the
                // trie.
                // If found, returns the
                // Object[] data array
                // associated.
                // Else, returns null
                // db is the ObjectContainer
                // holding the trie
                public Object[] search(string
                ➥_key,
                                        Object
➥Container db)
                {
                        TriePnode t;
                        char c;
                        int index=0;
                        // Empty trie?
                        if((t=root)==null)
                        ➥return(null);
                        int slen = _key.Length;
                        for(int i=0; i<slen;
                        ➥i++)
                        {
                                c = _key[i];
                                if((index=
➥t.isCharOnNode(c))==-1)
return(null);
                                if(i==slen-1)
➥break;

db.activate(t,2);
                                t = t.getPnode
➥Pointer(index);
                        }
                        // Get the data
                        db.activate(t,3);
                        return(t.get
➥DnodePointers(index).getData());
                }
        }
}
```

```
// Open/create the database file
ObjectContainer db = Db4o.openFile
➥(databaseFilePath);

// Create an empty Binary tree, and an empty trie
BinaryTree mybintree = new BinaryTree();
Trie mytrie = new Trie();

// Sit in an endless loop, reading text,
//  building objects, and putting those objects
//  in the database
while(true)
{
  // Read a word.
  // If we read a "#", then we're done.
  theword = ReadWord();
  if(theword.Equals("#")) break;

  // Read the pronunciation and put
  //  it in the object
  pronunciation = ReadPronunciation();
  _dictEntry = new DictEntry(theword,
  ➥pronunciation);

  // Read the number of definitions
  numdefs = ReadNumOfDefs();

  // Loop through definitions. For each,
  //  read the part of speech and the
  //  definition, add it to the definition
  //  array.
  for(int i=0; i<numdefs; i++)
  {
    partofspeech = ReadPartOfSpeech();
    definition = ReadDef();
    Defn def = new Defn(partofspeech,
    ➥definition);
    _dictEntry.add(def);
  }
  // We've read all of the definitions.
```

© Kiskapu Kft. Minden jog fenntartva

© Kiskapu Kft. Minden jog fenntartva

```
  // Put the DictEntry object into the
  //  database
  db.set(_dictEntry);

  // Now insert _dictEntry into the binary tree
  //  and the trie
  mybintree.insert(_dictEntry.TheWord,
  ➥_dictEntry);
  mytrie.insert(_dictEntry.TheWord, _dictEntry);
}

// All done.
// Store the binary tree and the trie
db.set(mybintree);
db.set(mytrie);

// Commit everything
db.commit();
```

Mindehhez persze szükség van néhány kisegít metódusra, amelyek például felolvassák a forrásfájlt, de a dolog logikája a fenti kódból azt hiszem világos. Figyeljük meg újra, hogy valamennyi indexelemet a teljes valójában képesek voltunk letárolni úgy, hogy egyszerűen csak átadtuk a megfelelő gyökérelemet a `db.set()` metódusnak. Az egész tehát egyetlen hívással megoldható!
Ha le akarunk kérdezni valamit az adatbázisból, az már kicsit trükkösebb. Itt már nem kezelhetjük a korábbihoz hasonló mértékben azonos módon az adatbázis objektumokat és az átmeneti adatszerkezeteket. A lemezen tárolt objektumokat be kell olvasni a memóriába, ez pedig egy explicit utasítást igényel. Az első lépés persze most is a `db.get()` metódus meghívása, amivel kikeressük az index gyökérelemét. Akár bináris fát, akár a másik módszert használjuk is az indexelésre, egy szó kikeresése a következőképpen fest kódszinten:

```
public static void Main(string[] args)
{
  Object[] found;
  DictEntry _entry;

  // Verify proper number of arguments
  if(args.Length !=3)
  {
    Console.WriteLine("usage: SearchDictDatabase
    ➥<database> B|T <word>");
    Console.WriteLine("<database> = path to db4o
    ➥database");
    Console.WriteLine("B = use binary tree; T =
    ➥use trie");
    Console.WriteLine("<word> = word to search
    ➥for");
    return;
  }

  // Verify 2nd argument
  if("BT".IndexOf(args[1])==-1)
  {
    Console.WriteLine("2nd argument must be B or
    ➥T");
    return;
  }

  // Open the database file
  ObjectContainer db = Db4o.openFile(args[0]);
  if(db!=null) Console.WriteLine("Open OK");

  // Switch on the 2nd argument (B or T)
  if("BT".IndexOf(args[1])==0)
  { // Search binary tree
    // Create an empty binary tree object for the
    //  search template
    BinaryTree btt = new BinaryTree();
    ObjectSet result = db.get(btt);
    BinaryTree bt = (BinaryTree) result.next();

    // Now search for the results
    found = bt.search(args[2],db);
  }
  else
  { // Search trie
    // Create an empty trie object fore the
    // search
    //  template
    Trie triet = new Trie();
    ObjectSet result = db.get(triet);
    Trie mytrie = (Trie) result.next();

    // Now search for the results
    found = mytrie.search(args[2],db);
  }

  // Close the database
  db.close();

  // Was it in the database?
  if(found == null)
  {
    Console.WriteLine("Not found");
    return;
  }

  // Fetch the DictEntry
  _entry = (DictEntry)found[0];
... <Do stuff with _entry here> ...
```

És akkor most elérkeztünk arra a pontra, amikor megindokolhatjuk, miért is használtuk azokat a bizonyos `db.activate()` hívásokat a keresési metódusokban az 1. és 3. Listában egyaránt.
Amikor meghívjuk a `db.set()` metódust, akkor – amint azt korábban is említettem – a db4o motor automatikusan végigpásztázza a megadott objektum valamennyi függőségét, végighalad az objektumfán és állandóvá tesz valamennyi abban elérhető elemet. Ezt a módszert egyébként ***elérhetőség alapján történő véglegesítésnek*** (***persistence by reachability***) hívják. A fordított irányban ugyanakkor, vagyis amikor a `db.get()` metódust hívjuk meg, hogy lekérjünk vele egy tárolt objektumot,

a *db4o* nem emeli be a teljes objektumfát. Ha ezt tenné, abból az következne, hogy amikor lekérjük mondjuk egy index gyökérelemét, akkor a rendszer a teljes indexet végigolvasná, plusz az összes szótárbejegyzést, plusz az összes meghatározást, és mindezeket beemelné a memóriába. Talán nem nehéz belátni, hogy egyetlen szó kikeresésének nem éppen ez a legcélravezetőbb módja.

Ehelyett a db4o az úgynevezett aktiválási mélység koncepciót használja. Tegyük fel, hogy a `db.get()` metódushívással lekértük az A objektumot, ami így bekerült a memóriába. Ha ezután meghívjuk a `db.ativate(A,6)` metódust, akkor ezzel azt mondjuk a *db4o* motornak, hogy az `A`-t tartalmazó fából szükségünk lesz minden gyermekobjektumra is, mégpedig hatszoros mélységig. Az indexek kezelése során felbukkanó `db.activate()` hívások tehát csupán arról gondoskodnak, hogy mindig bekerüljön a memóriába a szükséges adatmennyiség, amivel a keresés folytatható. (A keresés végén pedig hasonló módon jutunk hozzá a megtalált szótárelemekhez.)

## Egyedi indexek

Az objektum-orientált adatbázisok olyan flexibilitást adnak a fejlesztő kezébe, amit az *RDBMS* rendszerekből nem könnyen lehet kicsikarni. Különösen akkor mutatkozik meg ez, ha mély, összetett adatszerkezeteket tervezünk. Ilyenkor csak annyi a dolgunk, hogy a megfelelő hívással letároljuk, állandóvá tesszük a létrehozott objektumot, azzal pedig nem kell törődnünk, miként lehet összeegyeztetni az objektummodellt a relációs modellel.

Ami a cikkben bemutatott db4o motort illeti, ez egy olyan objektum-központú adatbázis-kezelő, ami ráadásul abban sem gátol meg bennünket, hogy egyszerre alakítsuk ki magát az adattartalmat, és a hozzá tartozó indexet. Bár a most bemutatott bináris fa és „gyöngysor” (trie) index nem volt különösebben bonyolult, annak bemutatására kiválóan megfeleltek, hogy a fejlesztő adott esetben ennél sokkal összetettebb indexelési és navigációs szerkezeteket is könnyűszerrel megvalósíthat. A tervezés során tehát létrehozzuk az alkalmazásnak legmegfelelőbb adatszervezési struktúrát, a megvalósítás során pedig nyugodtan dolgozhatunk a jó öreg objektumokkal, akár *Java*-t, akár a *Mono*/*.NET* párost használjuk. A legjobb az egészben azonban az, hogy a *db4o* nyílt forrású, így semmi meg nem akadályozhat bennünket abban, hogy fölhasználjuk a legközelebbi fejlesztésünkhöz. Aki többre kíváncsi, az látogasson a ➲ *www.db4objects.com* webhelyre.

*Linux Journal 2006., 142. szám*

*Rick Grehannak* több cikke jelent már meg a Byte, Embedded Systems Journal, JavaPro, InfoWorld, és a Microprocessor Report című lapokban. Társszerzője három könyvnek. Az egyik a távoli eljáráshívásról szól, egy másik a beágyazott rendszerekről, a harmadik pedig az objektum-orientált Java adatbázisokról. Jelenleg minőségbiztosítási vezető a Compuware NuMega Labs-nél.

© Kiskapu Kft. Minden jog fenntartva