Programozás III

RENDEZETT JLIST

LIST vs MODELL

Létrehozunk List-et is és modellt is, és oda-vissza rakosgatjuk az elemeket.

Miért nem jó, ill. mikor jó, mikor nem?

Ez jobb?

Object[] tomb = modell.toArray(); vagy

List<Tipus> temp =Collections.list(modell.elements());

– és ezt rendezzük, majd visszaírjuk;

Valamivel jobb, de nem tökéletes.

JLIST ELEMEINEK RENDEZÉSE

Ha rendezetten akarjuk kiíratni egy JList elemeit, akkor a modellt kell rendezni.

A DefaultListModell-nek nincs sort() metódusa. ☹

Saját modellt kell írni.

(Általában: egy kicsit is érdekesebb feladathoz illik saját modellt írni, vagyis ez nem nagy igény.)

```
public class DiakModell<E extends Diak> extends AbstractListModel{

    private List<E> diakLista;

    public DiakModell() {
        super();
        this.diakLista = new ArrayList<>();
    }

    @Override
    public int getSize() {
        return diakLista.size();
    }

    // Ezek kötelezően implementálandó metódusok
    @Override
    public Diak getElementAt(int index) {
        return diakLista.get(index);
    }

    public void addDiak(E obj) {
        this.diakLista.add(obj);
        int ujIndex = diakLista.size()-1;
        this.fireIntervalAdded(obj, ujIndex, ujIndex);
    }
```

SAJÁT MODELL

```
public void addDiak(E obj) {
    this.diakLista.add(obj);
    int ujIndex = diakLista.size()-1;
    this.fireIntervalAdded(obj, ujIndex, ujIndex);
}
// A fireIntervalAdded() metódus értesíti a felületet arról, hogy
// változás történt. Ugyanezt teszi a fireIntervalRemoved() metódus is.

public void removeDiak(E obj){
    int toroltIndex = diakLista.indexOf(obj);
    this.diakLista.remove(obj);
    this.fireIntervalRemoved(obj, toroltIndex, toroltIndex);
}
```

stb…

JLIST ELEMEINEK RENDEZÉSE

Az „érdekesebb” feladatoknál inkább angol elnevezéseket szokás használni, $\Rightarrow$ most is ilyen elnevezések lesznek:

```
public class SortableListModel<T extends Comparable<T>>
                        extends AbstractListModel {

    private List<T> model = new ArrayList<>();
    private boolean isSorted = false;

   // Kötelező implementálni a köv. két metódust (generálható):
    @Override
    public Object getElementAt(int index) {
        return model.get(index);
    }

    @Override
    public int getSize() {
        return model.size();
    }
```

JLIST ELEMEINEK RENDEZÉSE

A SortableListModel további metódusai:

```
/**
 * Ha utólag akarjuk rendezni a modellt, akkor ezt
 * a metódust kell hívni, de csak akkor hajtódik végre,
 * ha még nincs rendezve.
 */
public void sort() {
    if (!isSorted) {
        Collections.sort(model);
        fireContentsChanged(this, 0, model.size() - 1);
    }
}
```

Minden változásról értesíteni kell a JList-et.
(a DefaultListModel-ben is van ilyen)

JLIST ELEMEINEK RENDEZÉSE

A SortableListModel további metódusai:

```
/**
 * a modell végére szúr be egy elemet
 * @param element
 */
private void addElement(T element) {
    this.addElement(element, model.size());
    this.fireIntervalAdded(element, model.size()-1, model.size()-1);
}

/**
 * a modell adott indexű helyére szúrja be az elemet
 * @param element
 * @param index
 */
private void addElement(T element, int index) {
    model.add(index, element);
    this.fireIntervalAdded(this, index, index);
}
```

```java
/**
 * Ha sort értéke true, akkor már eleve rendezetten rakja be az elemet,
 * vagyis az eddig berakottakat is rendezi, és ennek megfelelő helyre
 * rakja az újat.
 * Ha a sort értéke false, akkor a lista végére teszi az új elemet.
 * @param element
 * @param sort
 */
public void addElement(T element, boolean sort) {
    if (!sort) {
        addElement(element);
        isSorted = false;
    } else {
        if (!isSorted) {
            sort();
        }
        int index = Collections.binarySearch(model, element);
        System.out.println("index " + index);
        if (index < 0)
            addElement(element, -index - 1);
        else
            addElement(element, index);
        isSorted = true;
    }
}
```

JLIST ELEMEINEK RENDEZÉSE

„Magyarított” és kommentezett változatát ld.
witch (coospace) …/eloadasok/peldak

JLIST ELEMEINEK RENDEZÉSE

A modellben a bináris rendezést használtuk.

Ennek Java megvalósításához ld. rekurzió.

MOTIVÁCIÓ-FÉLE

Részlet egy állásinterjúkról szóló levélből:

A programozós kérdések rendszerint ugyanazokra a területekre térnek ki:

1. Néhány alapvető kérdés a JRE, JVM, JDK környékéről. Ezek többnyire egyszerűek, bár előfordulhat nehezebb memóriakezeléssel kapcsolatos is.

2. Algoritmus-elméletet nem mindenhol, de kérdezhetnek: Rendezések, keresések stb. **Egy bináris keresést elmagyarázni angolul, illetve megírni Javaban nem is olyan könnyű...**

MOTIVÁCIÓ-FÉLE

Másik részlet (2014. okt. 7.):

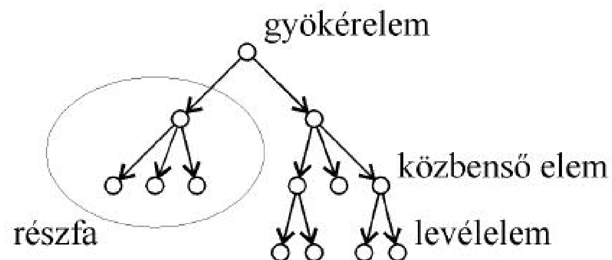
3. feladat: adott az alábbi interface. Ennek megfelelően olyan metódust kell írni, mely adott root Node-ból indulva bejárja a fát, és kiválasztja a legnagyobb értékű Node-ot. Erre negyed óra volt, rekurzióval kellett megoldani.

```
interface MyNode {
    List<Node> getChildren();
    int getValue();
}
```

HIERARCHIKUS ADATSZERKEZETEK

FA (tree)

Dinamikus, homogén adatszerkezet, amelyben minden elem megmondja a rákövetkezőjét.



HIERARCHIKUS ADATSZERKEZETEK – BINÁRIS FA

BINÁRIS FA:

A számítástechnikában kitüntetett szerepe van a bináris fának.
A bináris fa olyan fa, amelyben minden elemnek legfeljebb két rákövetkezője lehet.
Szigorú értelemben vett bináris fáról akkor beszélünk, amikor minden elemnek 0 vagy pontosan 2 rákövetkező eleme van.

Létrehozása:
Létrehozzuk az üres fát, majd létrehozzuk a gyökérelemet, ezután bővítjük a fát.

HIERARCHIKUS ADATSZERKEZETEK – BINÁRIS FA

Speciális faművelet a **bejárás**:
A bejárás az a tevékenység, amikor a fát, mint hierarchikus adatszerkezet egy lineáris adatszerkezetre képezzük le.
A fa elemeit a bejárás folyamán egyszer, és pontosan egyszer érintjük, az elemek között valamilyen sorrendet állapítunk meg, attól függően, hogy hogyan járjuk be a fát.

Bejárási módok megkülönböztetése: attól függően, hogy mikor kerül sorra a gyökérelem.

HIERARCHIKUS ADATSZERKEZETEK – BINÁRIS FA

Bejárások:

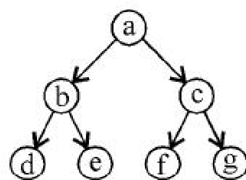
Preorder bejárás: Ha a fa üres, akkor vége a bejárásnak, egyébként vegyük a **gyökér**elemet és dolgozzuk fel. Ezután járjuk be preorder módon a **bal**oldali, majd a **jobb**oldali részfát.

Inorder bejárás: Ha a fa üres, akkor a bejárás befejeződik. Egyébként járjuk be inorder módon a **bal**oldali részfát, majd dolgozzuk fel a **gyökér**elemet, és járjuk be ugyanezen módszerrel a **jobb**oldali részfát.

Postorder bejárás: Üres fánál vége, egyébként járjuk be postorder módon a **bal**oldali majd a **jobb**oldali részfát, végül dolgozzuk fel a **gyökér**elemet.

Megvalósítás: rekurzióval

HIERARCHIKUS ADATSZERKEZETEK – BINÁRIS FA



preorder: (gyökér, bal, jobb) – abdecfg

inorder: (bal, gyökér, jobb) – dbeafcg

postorder: (bal, jobb, gyökér) – debfgca

Bináris rendezés: ügyes faépítés + megfelelő bejárás

HIERARCHIKUS ADATSZERKEZETEK – FA

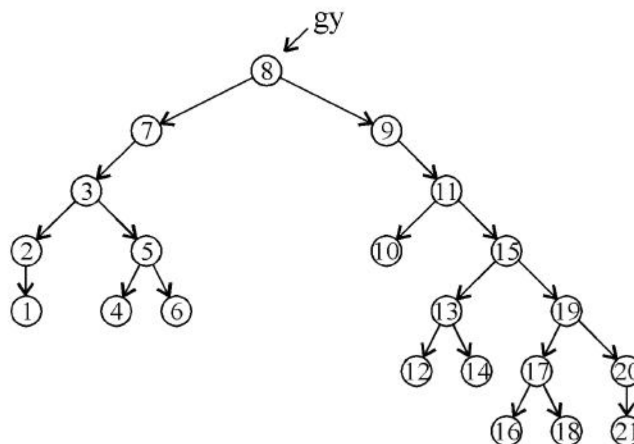
KERESŐ-FA:

A bináris fákat gyakran használják olyan adathalmaz feldolgozására, amikor az adatelemeknek van egy kulcsrészük (táblázatok) vagy maguk az adatelemek különböző értékűek. Ilyenkor a kulcs a feldolgozás alapja.

Ha adott elemszám mellett úgy építjük fel a fát, hogy bármely elemére igaz, hogy az elem baloldali részfájában az összes elem kulcsa kisebb, a jobboldali részfájában az összes elem kulcsa pedig nagyobb az adott elem kulcsánál, akkor **keresőfá**ról (vagy rendezőfáról) beszélünk.

HIERARCHIKUS ADATSZERKEZETEK – FA

Kereső-fa – példa:



HIERARCHIKUS ADATSZERKEZETEK – FA

A keresőfa jelentősége:
Ha egy adott elemet meg akarunk keresni a fában, akkor a gyökértől kiindulva bármelyik kulcsú elem megkereshető úgy, hogy vizsgáljuk: a gyökérelem kulcsa megegyezik-e a keresett elem kulcsával.
Ha igen, megállunk, ha nem, megnézzük, hogy annál kisebb-e vagy nagyobb.
Ha nagyobb, akkor a jobboldali részfán, ha kisebb, akkor a baloldali részfán haladunk tovább ugyanezzel a módszerrel mindaddig, míg az elemet meg nem találjuk, vagy nem jutunk el egy olyan elemhez, amelyiknél az adott irányban nincs több elem, ekkor a keresett kulcsú elem nem szerepel a fán.

HIERARCHIKUS ADATSZERKEZETEK – FA

A keresés gyors, mert maximum a fa magassága+1 hasonlítással vagy megtaláljuk az elemet, vagy az elem nincs a fában.
A keresőfában az elemek olyan sorrendben vannak, hogy ha inorder módon járjuk be a fát, akkor az elemek egy rendezett sorozatát kapjuk.

HIERARCHIKUS ADATSZERKEZETEK – FA

A keresőfa létrehozása:

Vegyük az elemeket az adott sorrendben.
Az első elem lesz a fa gyökere.
A második elemet véve nézzük meg, szerepel-e már a fában.
Ha igen, akkor hiba történt (kétszer nem lehet benne ugyanaz az elem), ha nem, akkor döntünk, hogy az hol helyezkedjen el az előzőhöz képest (kisebb vagy nagyobb a gyökérelemnél).
Ha nagyobb, akkor a jobboldali részfának lesz az eleme, ha pedig kisebb, akkor a baloldalinak.
Beillesztjük az új elemet a részfába: addig megyünk, amíg levélelemet nem találunk, mert mindig levélelemmel bővítünk.

BINÁRIS FA KEZELÉSE – REKURZIÓVAL

Bináris fa definíciója:

```
public class BinarisFa {
    private BinarisFa bal;
    private BinarisFa jobb;
    private int gyoker;

    public BinarisFa(int gyoker) {
        this.gyoker = gyoker;
    }
```

+ set/get

```java
/* Példa fa-struktúra felépítésére, bejárására */

public class FaDemo {

    public void indit() {

        int i;
        int tomb[] = new int[]{5, 1, 8, 6, 3, 9};
        System.out.println("A számok eredeti sorrendje: ");

        for (i = 0; i < tomb.length; i++) {
            System.out.println(tomb[i]);
        }

        BinarisFa fa = new BinarisFa(tomb[0]);

        for (i = 1; i < tomb.length; i++) {
            beszur(fa, tomb[i]);
        }

        System.out.println("\nA fa létrehozása utáni sorrend:");
        bejar(fa);
    }
```

BINÁRIS FA KEZELÉSE – REKURZIÓVAL

```java
public void beszur(BinarisFa fa, int adat) {
    if (adat < fa.getGyoker()) {
        if (fa.getBal() != null) {
            beszur(fa.getBal(), adat);
        } else {
            System.out.println("Beszúrtam " + adat + "-t "
                    + fa.getGyoker() + " bal oldalára");
            fa.setBal(new BinarisFa(adat));
        }
    } else {
        if (fa.getJobb() != null) {
            beszur(fa.getJobb(), adat);
        } else {
            System.out.println("Beszúrtam " + adat + "-t "
                    + fa.getGyoker() + " jobb oldalára");
            fa.setJobb(new BinarisFa(adat));
        }
    }
}
```

BINÁRIS FA KEZELÉSE – REKURZIÓVAL

```
public void bejar(BinarisFa fa) {
    if (fa != null) {
        bejar(fa.getBal());
        System.out.println(fa.getGyoker());
        bejar(fa.getJobb());
    }
}


public class BinarisRendezes {

    /**...*/
    public static void main(String[] args) {
        new FaDemo().indit();
    }
}
```

BINÁRIS FA KEZELÉSE – REKURZIÓVAL

```
A számok eredeti sorrendje:
5
1
8
6
3
9
Beszúrtam 1-t 5 bal oldalára
Beszúrtam 8-t 5 jobb oldalára
Beszúrtam 6-t 8 bal oldalára
Beszúrtam 3-t 1 jobb oldalára
Beszúrtam 9-t 8 jobb oldalára

A fa létrehozása utáni sorrend:
1
3
5
6
8
9

Process completed.
```