CODECOOL

5 népszerű IT interjúkérdés válaszokkal

Bevezető

Nem könnyű kiválasztani öt kérdést, ami valóban jellemző az IT szakmai interjúkra. Ahány interjúztató, annyiféle vélemény és preferencia. Különböző projekteken vagypozíciókban más-más kvalitások lehetnek fontosak. Gyakran már az első kérdésre adott válasz meghatározza, mi legyen a következő kérdés. Előfordulhat, hogy , elsőre nem volt elég meggyőző a jelölt válasza, és az interjúztatók kíváncsiak a gondolkodásmódjára, hiszen előfordulhat,, hogy épp csak rossz oldalról közelítette meg a témakört. Van, hogy azért, mert mondott valamit, ami felkeltette az interjúztatók érdeklődését, ami jól megmutatja a jelölt készségeit, és ezt akarják jobban feltárni.

Kiválasztottunk öt olyan kérdést, amelyek más-más szinten mennek a szakmai részletekbe, illetve különböző technikai mélységűek. Emellett törekedtünk arra, hogy ne legyenek nyelv-, vagy technológia specifikusak. Persze egy éles helyzetben lesznek kérdések, amik ezeknél sokkal specifikusabbak, a betöltendő pozícióhoz kapcsolódóan. A kérdések célja az, hogy képet kapjunk a jelentkező tudásáról, nem pedig az, hogy elhangozzon egy lexikonba illő definíció.

Minden kérdéshez írunk egy lehetséges választ is. Fontos megjegyezni, hogy nincs egyértelműen helyes megoldás. A válaszunkhoz adunk egy kis magyarázatot is, ahol lényegre törően belemegyünk a fontos aspektusokba.

Mikor használnál listát és mikor halmazt(set)?

1

Ilyenkor sokszor az dönt, hogy milyen műveletet fogok gyakran csinálni.

Halmazt akkor használunk, amikor duplikációkat szeretnénk kiszűrni. Vagy akkor, ha azt akarjuk ellenőrizni, hogy egy elemet láttunk-e már, illetve hányféle különböző elemet láttunk eddig.

Listát akkor alkalmazunk, amikor fontos a sorrendiség, illetve amikor gyakran akarjuk összefűzni vagy szétbontani a listákat.

Ha az a rendszeres, hogy minden elemen végig akarunk menni, akkor a sorrendiség dönt. Ha fontos az általunk meghatározott mód, akkor listát használnánk, ha nem számít, akkor inkább halmazt. Továbbá vannak halmaz implementációk – például a TreeSet –, amelyek megtartanak egy természetes sorrendiséget (például ábécésorrend).

Itt fontos tudni azt is, hogy a tipikus műveleteknek mi az időkomplexitása. De ezt csak megemlíteni javasoljuk, ne vesszünk el a részletekben, az eredeti kérdés nem erre irányult. Ha az interjúztató kíváncsi erre is, akkor van rá lehetősége, hogy jobban belemenjen egy következő kérdés feltevésekor.

Hasonlóan fontos megjegyezni, hogy ezeknek az adatstruktúráknak többféle megvalósítása is létezik. Ezt is érdemes megemlíteni, és egy későbbi beszélgetésben akár részletezhetjük is.

Mi az a design pattern? Tudsz példákat mondani?

2

Visszatérő problémákra visszatérő megoldások, amelyekre nincsen dedikált nyelvi elem, ezért létező nyelvi elemek kötött kombinációját használjuk.

Három külön kategóriába soroljuk őket:

Creational Példányok létrehozását segíti, például builder

Structural Nagyobb logikai struktúrák létrehozását segíti flexibilis módon, például *decorator*

Behavioral Különböző-, vagy fix szerepű egységek rendszerezését segítik, például *visitor*

Rengeteget lehet erről is beszélgetni. Fontos, hogy nem szükséges az összeset fejből tudni. Jó, ha felismerjük, hogy egy olyan kihívással szembesültünk, amire valószínűleg van ismert megoldás, amit meg is tudunk keresni, és a leírás alapján alkalmazni. Ebben a helyzetben egy junior és egy senior szakember között már csak az a különbség, hogy ezt a folyamatot hányszorcsinálta végig. :)

Ami itt még izgalmas, hogy ahogy fejlődnek a programnyelvek, egyre kevesebbre design pattern-re van szükség. Azokban a nyelvekben, ahol van foreach stílusú loop, nincs szükség az iterator patternre. Azokban a nyelvekben meg--mint például a **Clojure**--ahol van multiple dispatch, ott nincs szükség a *visitor pattern*-re.

Egy tömbben minden szám 1-től 100-ig egyszer szerepel, kivéve egyet, ami egyszer sem. Hogy deríted ki, melyik a hiányzó szám?

3

Kiszámolhatjuk, mennyi lenne mind a 100 szám összege a számtani sor összegképletét használva, ahol n-t 100-nak választva 5050-et kapunk . $\frac{(n+1)*n}{2}$

Összeadva a tömb minden elemét, kapunk egy számot, ami pont az eredmény értékével különbözik a fent kiszámolt 5050-től.

A fenti megoldás előnye, hogy minden számot, csak egyetlen egyszer vizsgálunk meg, és a felhasznált memória is fix, csak azt kell megőrizni, hogy mi az eddig összeadott számok összege.

Az összes szám ellenőrzése is elkerülhető, ha rendezett a tömb. Viszont a rendezés maga több idő, mint csak egyszer simán végigmenni a tömbön. Viszont az általános megoldás mellé–nem helyette–megadhatunk megoldást a rendezett esetre is, ha van rá érdeklődés.

Itt ha nem tudunk azonnal egy optimális megoldást, akkor se csüggedjünk. Teljesen érvényes közösen ötletelni, amíg azt mi vezetjük. És ha van egy működő megoldásunk és el is tudjuk mondani, mik a potenciális hátulütői, költségei, az is egy értékes válasz.

Mi a különbség a statikusan és dinamikusan tipusos nyelvek között?

4 Minden program tartalmazhat olyan hibákat ahol rossz formájú adaton próbálunk elvégezni egy műveletet. Például nincs értelme összeadni egy banánt egy naranccsal, vagy megnézni, hány elemet tartalmaz egy *igaz* érték.

Dinamikusan tipusos nyelveknél ezzel akkor szembesülünk, amikor a futó program megpróbálja végrehajtani a műveletet a hibás adaton, statikus tipusos nyelveknél viszont még a program futtatása előtt van lehetőségünk csinálni egy ellenőrzést, ami az ilyen fajta hibák nagy részét megtalálja anélkül, hogy a programot futtatni kellene.

Gyakori hiba itt, hogy sokan keverik a gyengén/erősen tipusos megkülönböztetést a dinamikus/statikussal, ami két különböző nézőpont szerinti különválasztás.

Sokan hiszik azt is, hogy statikusan tipusos nyelveknél szükséges beleírni a programba az értékek tipusait. Most már **C++**-ban, **C#**-ban és **Java**-ban is van tipus inferencia, amik az első statikus tipusos nyelvek, amikkel kezdő programozók találkoznak.

Tovább árnyalja a kérdést, hogy például **Python**-ban van lehetőség statikus tipus ellenőrzésre külső eszközök segítségével.

Micsoda egy verziókövető rendszer?

5

Egy program, aminek a segítségével a forráskód változásait követjük eszközöljük.

A csapatban történő fejlesztést nagyban segíti, emellett a különböző **DevOps** rednszerek is támaszkodnak rá.

Két nagy kategóriájuk a centralizált és decentralizált verziókövetők.

Példák:
Centralizált *subversion, CVS, ClearCase*
Decentralizált *git, mercurial, darcs*

Ma a de-facto program a *git*. Régebbi projektek esetén könnyen találkozhatunk más rendszerekkel is. Aki az alapnál picit jobban ért a GIT-hez, hamar a csapat hasznos tagjává válik. Fontos érteni, hogy micsoda egy *merge conflict* és hogyan lehet azt feloldani.

Ha ennél többet akarsz tudni az IT szakmai interjúkról, akkor jelentkezz a következő IT karrier fejlesztő képzésünkre!