**Adatszerkezetek és algoritmusok**

**Nagy kérdések**

**Algoritmus, mint programozói eszköz:** Informálisan algoritmusnak nevezünk bármilyen jól definiált számítási eljárást (elemi lépések sorozatát), amely bemenetként bizonyos értéket vagy értékeket kap és kimenetként bizonyos értéket vagy értékeket állít elő. Eszerint az algoritmus olyan számítási lépések sorozata, amelyek a bemenetet átalakítják kimenetté. Az algoritmusokat tekinthetjük olyan eszköznek is, amelynek a segítségével pontosan meghatározott számítási feladatot oldunk meg. Ezeknek a feladatoknak a megfogalmazása általában a bemenet és a kimenet közötti kívánt kapcsolat leírása. Kicsit általánosabban: Olyan eljárás (elemi lépések sorozata), melynek során a következők teljesülnek:

* Jól meghatározott objektumokon jól meghatározott műveleteket végzünk
* Minden lépés elvégzése után egyértelműen definiált helyzet áll elő
* Véges sok lépés után véget ér
* Nem csak egy feladatra, hanem egy feladatosztály tagjaira érvényes.

Megadhatók természetes, emberi nyelven, pontokba szedett természetes nyelvi utasításokkal, folyamatábrával, pszeudonyelvvel, valamilyen programozási nyelven. Tartalmazhat szekvenciákat (utasítások végrehajtása felírás sorrendjében), szelekciót (elágazások), iterációt (ciklusok), alprogramok hívását.

**Beszúró rendezés:** A beszúró rendezés eljárás bemutatását az utasítások időköltségével és az egyes utasítások végrehajtásainak számával kezdjük. Minden j=2,3…n-re, ahol n=méret[A], tj legyen az a szám, ahányszor a while ciklus 4. sorban lévő tesztje végrehajtódik az adott j értékre. Ha egy for vagy while ciklusból a szokásos módon lépünk ki, akkor a teszt eggyel többször hajtódik végre, mint a ciklusmag. Feltesszük, hogy a megjegyzések nem végrehajtható utasítások, így nem vesznek igénybe időt. Ekkor az első utasítás n, a 2., 3., és 8. utasítás n-1, a többi pedig ∑tj alkalommal futnak le. Ezeket összeadva és megszorozva ci-vel kapunk egy egyenletet. Legjobb esetben a tj mindenhol 1, akkor behelyettesítve azt az eredményt kapjuk, hogy a\*n+b, tehát T(n) lineáris függvénye n-nek. Amennyiben tj =j, akkor az eredményünk a\*n2+b\*n+c, ekkor T(n) négyzetes függvénye n-nek. Átlagos esetben tj =j/2, ekkor az eredmény a\*n2+b\*n+c, ami szintén négyzetes függvény. C1n2.

**Összefésülő rendezés:** Az összefésülő rendezés algoritmusa szorosan követi az oszd-meg-és-uralkodj paradigmát. Ez n elemű rendezendő sorozatot felosztja két n/2 elemű részsorozatra, a két részsorozatot pedig összefésülő rendezéssel rekurzív módon rendezi. Ezután összefésüli a két részsorozatot, létrehozva a rendezett választ. Annak belátásához, hogy az összefésül algoritmus θ(n) ideig fut, ahol n=r-p+1, figyeljük meg, hogy az 1-3. és a 10-11. sorok állandó időt vesznek igénybe, a 4-9. sorokban szereplő for ciklusok θ(n1+n2)=θ(n) időt, a 12-19. sorokban szereplő for ciklusnak pedig n iterációja van, és mindegyik állandó ideig tart. Amikor egy algoritmus rekurzívan hívja magát, futási idejét gyakran rekurzív egyenlőséggel vagy rekurzióval írhatjuk le, amely az n méretű feladat megoldásához szükséges teljes futási időt a legkisebb bemenetekhez tartozó futási időkkel fejezi ki. A rekurzió megoldására matematikai eszközöket használunk, melyekkel az algoritmusra teljesítménykorlátokat tudunk adni.

Ha n<c, akkor T(n)=θ(1), egyébként T(n)=2\*T(n/2)+D(n)+C(n). Esetünkben könnyű belátni, hogy D(n) € θ(1) és már láttuk, hogy C(n) € θ(n). Így tehát ha n<c, akkor T(n)=θ(1), egyébként T(n)=2\*T(n/2)+θ(n). Ebből az iteratív feltételezésből következik, hogy: T(n) € θ (n\*log2\*n). C2\*n\*log2n.

**Helyettesítő módszer**: A rekurziók helyettesítő módszerrel való megoldása két lépésből áll: sejtsük meg a megoldást, és a teljes indukcióval határozzuk meg az állandókat és igazoljuk a megoldás helyességét. Az elnevezés onnan ered, hogy a helyesnek vélt megoldást be kell helyettesíteni a függvénybe, miközben indukciós feltevést kisebb értékekre alkalmazzuk. Csak akkor alkalmazható, ha a helyes válasz könnyen megsejthető. A helyettesítő módszer rekurzióval megadott függvény felső vagy alsó korlátjának a meghatározására is használható. Határozzuk meg a T(n)=2T\*(└n/2┘)+n rekurzív képlettel jellemzett függvény felső korlátját. Sejtésünk megoldása az, hogy T(n)=O(n|g n). Módszerünk az, hogy bebizonyítjuk megfelelően választott c>0 állandóra T(n)<cn|g n. Először is feltesszük, hogy ezzel a korláttal k0<k<n-re érvényes az egyenlőtlenség, azaz T(k)<c └k┘ lg (└k┘). Ezt a rekurzív egyenletbe helyettesítve kapjuk, hogy (kihasználjuk, hogy n/2 <n): T(n)=cn lg n-cn lg2+n=cnlgn-cn+n≤cnlgn (ha c≥1). Most már csak n=1-re kell belátni, csakhogy c\*1\*lg1=0 ami nem nagyobb egyenlő T(1).  
Sajnos nincs általános szabály a megoldás megsejtésére. A jó sejtés kitalálásához gyakorlatra és találékonyságra van szükség. Persze van néhány heurisztikus módszer, ami segít.

**Rekurziós fa módszer:** A rekurziós fa minden csúcsa egy részfeladatnak felel meg: a függvény kiértékelésekor végrehajtódó minden rekurziós híváshoz tartozik egy csúcs. Szintenként összegezzük a csúcsok költségét, majd az így kapott szintenkénti költségeket összegezzük, hogy megkapjuk a teljes költséget. Oszd-meg-és-uralkodj elvet használó algoritmusok elemzésénél a rekurziós fa nagyon kényelmes eszköznek bizonyul. A rekurziós fa módszert leginkább egy jó sejtés megtalálására érdemes használni, amit aztán a helyettesítő módszerrel ellenőrzünk. (Nézd meg a diát.)

**Mester módszer:** Ez a módszer receptet ad a T(n)=aT(n/b)+f(n) alakú rekurzív egyenletek megoldására, ahol a >1 és b>1 állandók, és az f(n) aszimptotikusa pozitív függvény. Szemléletesen, egy rekurzív eljárás az eredeti problémát felosztja a darab, egyenként n/b méretű részproblémára, a felosztás és az összevonás együttesen f(n) lépésben valósul meg. Belátható, hogy amennyiben n/b helyére valamelyik egészrészt írjuk az nem érinti a lényeget. A mester módszer alkalmazásához három esetet kell megjegyezni, ezután sok rekurzív egyenlet megoldása könnyen, sok esetben papír és ceruza nélkül meghatározható. (Dia.)

**Ω(n\*lg\*n):**

**Absztrakt adatszerkezetek osztályozása:** Adatszerkezetek elemeinek számának változása alapján: statikus, dinamikus. Elemeinek típusa szerint: homogén, heterogén. Adatelemek közötti kapcsolat: homogén adatszerkezetnél lehet struktúra nélküli, asszociatív, szekvenciális, hierarchikus, hálós, heterogénnél nem csoportosítjuk ilyen szempontok alapján.

**Absztrakt adatszerkezetekkel végezhető műveletek:** Létrehozás, módosítás (bővítés, törlés (fizikai, logikai), csere), rendezés, keresés, elérés, bejárás, feldolgozás.

**Adatszerkezetek reprezentációja:** Ábrázolás alatt az adatszerkezetek memóriában való megjelenési formáját értjük. Ez minden adatszerkezet esetén lehet folytonos (vektorszerű) vagy szétszórt (láncolt). Az adatelemek számára tárhelyeket foglalunk a memóriában, amely egy adatelem értékét tárolja, illetve szerkezetleíró információkat is hordozhat.

* Folytonos tárolás: Egy tárhelyen egy adatelem értékét tároljuk. A tárhelyek a memóriában folytonos, összefüggő tárterületet alkotnak, a tárhelyek mérete azonos. Előnye az, hogy közvetlen elérést ad, a kezdőcím és az egy adatelemhez tartozó tárhely méretének ismeretében, a csere művelete könnyen megvalósítható, hatékony rendező algoritmusoknál, kereső algoritmusoknál. Hátránya: nem segíti a bővítés és a fizikai törlés műveletének végrehajtását.
* Szétszórt tárolás: Egy tárhelyen egy adatelem értékét (adatrész) és legalább egy mutató értékét (mutatórész) tároljuk. A mutatók értékei memóriacímek lehetnek, amelyek megmondják az adatelem rákövetkezőinek tárbeli helyét. A tárhelyek mérete nem szükségképpen azonos, elhelyezkedésük a memóriában tetszőleges. Fajtái: egyirányba láncolt lista, cirkuláris lista, kétirányba láncolt lista, multilista.

Absztrakt adatszerkezetek grafikus képi megjelenítésénél használt jelölések: O adatelem, → kapcsolat két adatelem között. Amikor egy absztrakt adatszerkezethez megadjuk a tárolási módját és a leképezését, akkor megadjuk az absztrakt adatszerkezet reprezentációját. Ha a reprezentáció mellé megadjuk a műveletek megvalósítását (algoritmusok) is, akkor megadjuk az absztrakt adatszerkezet implementációját.

**Halmaz, multihalmaz:** A halmaz és a multihalmaz struktúra nélküli, homogén és dinamikus adatszerkezetek. A halmaz minden eleme különböző. A multihalmazban előfordulhatnak azonos elemek is. Mindkét adatszerkezetre igaz, hogy az adatszerkezetben lévő elemek között nincs kapcsolat.

* A halmaz: A halmaz adatszerkezet a matematikai fogalom megjelenése adatszerkezetek szintjén. Mindig véges – ennyiben nem felel meg teljesen a fogalomnak. Műveletei az eleme, unió, metszet, különbség. Vele végezhető hagyományos műveletek: létrehozás explicit módon (elemek felsorolása), vagy egy predikátum segítségével, bővítés unióképzéssel, törlés fizikai módon különbségképzéssel, csere nincs, a rendezés, csere, elérés, bejárás nem értelmezettek és a feldolgozás a halmaz alapműveleteinek segítségével. Reprezentációja folytonosan, karakterisztikus függvény segítségével történik.
* Multihalmaz: Abban különbözik a halmaztól, hogy megengedi az adatelemek ismétlődését. Műveletei: eleme, unió, metszet, különbség. A hagyományos műveletek végzése hasonló a halmazhoz. Multihalmaz feldolgozása alapműveletek segítségével történik. Reprezentációja folytonosan, karakterisztikus függvény segítségével történik.

**Tömb adatszerkezet:** Statikus, homogén és asszociatív adatszerkezet. A felépítése definiálja: benne az adatelemek egymáshoz viszonyított helyzete fontos. A tömb bármelyik eleme egész számok sorozatán keresztül érhető el. Minden adatelemhez különböző sorozat tartozik, így az asszociativitást biztosító részhalmazok egyértelműek és diszjunktak. A számsorozat számait indexeknek nevezzük, segítségükkel tudjuk az adatelemet kiválasztani. Az indexek darabszámát a tömb dimenziójának hívjük. Ha mást nem mondunk, akkor a tömb elemeinek indexelése mindegyik dimenzióban 1-től indul. Legegyszerűbb esete: egydimenziós tömb, vektor, de van kétdimenziós tömb: mátrix. Léteznek magasabb dimenziós tömbök is, tetszőlegesen nagy lehet, de mindig véges. **Műveletek:**

Létrehozás: rögzítjük a dimenziók számát és az indextartományokat. Ezzel egyben meghatározzuk a tömb elemszámát. A szerkezet kialakításával párhuzamosan elemeket helyezünk a tömbbe. Bővítés: nincs, mert a tömb statikus. Csere: bármely létező elem értékét felülírhatjuk egy új értékkel. Elhelyezhetünk elemet oda, ahová a létrehozáskor nem tettünk. Törlés: csak logikai. Elérés: közvetlen, indexekkel, rendezés értelmezhető, keresésé reprezentációfüggő, bejárás többdimenziósoknál reprezentációfüggő, feldolgozás alapja az elérés.

Háromszögmátrixok: Négyzetes, kvadratikus mátrixok, felső és alsóháromszög lehet. Felsőháromszögnél a főátlója alatt csupa 0 van, alsónál felette van csupa 0. Az értékes elemek száma sima négyzetes mátrixnál n2, háromszögnél (n\*(n+1))/2, ezért ennyi elemű V vektorra szoktuk leképezni.

Szimmetrikus mátrixok: Négyzetes mátrixok, melyre teljesül, hogy ai,j=aj,i.

Dinamikus tömb: Általában egydimenziós tömb, ezért lehet dinamikus vektor is. Mérete szűkebb értelemben feldolgozás során tetszőlegesen (dinamikusan) változik. Ebben az esetben ez gyakorlatilag egy szekvenciális lista. Tágabb értelemben fizikailag továbbra is statikus tömb, a létrehozáskor megadott elemszámot később viszont változtathatjuk. Ilyenkor a tömb végén lehetnek adatelemek által nem használt, létrehozáskor lefoglalt helyek. Bővítés tetszőleges helyen végrehajtható. Fizikai törlés bármely esetén értelmezhető. Műveletei megegyeznek a statikus tömbbel.

**Soros, önátrendező és** **rendezett táblázatok:**   
Soros táblázat: Az adatelemek táblázatbeli sorrendjét az elemek táblázatba kerülésének időbeli (érkezési) sorrendje, ill. az esetleges törlések sorrendje befolyásolja. Vele végezhető műveletek: létrehozás (kulcs és érték típusának meghatározása, és az elemek elhelyezése érkezési sorrendben), bővítés (tábl. végén), törlés (mind folytonos, mind szétszórt reprezentáció mellett fizikai törlés), csere (érték bármikor, kulcsérték akkor, ha az új érték még nem szerepel). Rendezés nem lehet, keresés kulcs alapján, elérés soros, bejárás elejétől végéig, feldolgozás alapja a kulcs és a teljes keresés. Folytonosan és szétszórtan (pl. egyirányba láncolt listával) egyaránt ábrázolható. Előbbi esetben fizikai törlést úgy valósíthatjuk meg, hogy a törlendő adatelemet a táblázat utolsó elemével felülírjuk, és csökkentjük a táblázat aktuális elemszámát. Utóbbi esetben mutatóérték cserével oldható meg.  
Önátrendező táblázat: Műveletek legtöbbje megegyezik a soros táblázatéval. Kivételt képez a feloldozás. Egy elem feldolgozás után a feldolgozott elemet a táblázat elejére helyezzük, az addigi első elem elé. Ennek következtében mindig a legutoljára feldolgozott elem lesz a táblázat elején. Nagyszámú adatelem feldolgozása után az elemek sorrendje a táblázatban jól fogja közelíteni a feldolgozási gyakoriságot: a táblázat elején lesznek a gyakrabban feldolgozott elemek. Az önátrendező táblázat legjobban szétszórtan, egyirányba láncolt listával reprezentálható. Ekkor a feldolgozás művelete három mutató értékének cseréjét jelenti.  
Rendezett táblázat: Elemei a kulcsértékek alapján rendezettek, az adatelemek sorrendjét a kulcsértékek (általában) növekvő sorrendje definiálja. Műveletek: A legtöbbje megegyezik a soros táblázatéval. Kivételt képeznek a következő műveletek: létrehozás és bővítés (érkező elemeket rendezetten helyezzük el a táblázatban, rendezett sorozatba történő beszúrással), csere (az adatelem értékrésze bármikor cserélhető), keresés (lineáris vagy bináris – reprezentációfüggő). Folytonos rep.-nál gyorsabb a keresés, de nehézkes a bővítés és a törlés. Szétszórtnál fordítva.

**Kulcstranszformációs táblázatok:** Véletlen szervezésű struktúrák. Célja: a szinte közvetlen elérés biztosítása a kulcsértékek alapján. Hash függvény: a kulcstranszformációs táblázatban egy K értékű kulccsal rendelkező elem helyét (címét) egy h függvény h(K) értéke határozza meg. Ezt a h függvényt hívjük hasító vagy hash függvénynek. Hashing: azt az eljárást, melynek során egy adatelem K értékű kulcsához meghatározzuk a h(K) (az adatelem táblázatbeli helyét), hasításnak, hashingnek, randomizálásnak vagy kulcstranszformációnak nevezzük. Ennek a táblázatfajtának az ábrázolása folytonos, vagy legalább a folytonos ábrázoláson alapul. Kölcsönösen egyértelmű hash függvény használható, ha a gyakorlatban előforduló kulcsértékek száma közel azonos az elvileg lehetséges kulcsértékek számával vagy a gyakorlatban előforduló kulcsértékek egyenletesen oszlanak el az elvileg lehetséges kulcsértékek között. Ha a gyakorlatban előforduló kulcsértékek száma és az elvi lehetőségek száma között nagy az eltérés, és a gyakorlatban előforduló kulcsértékeknek nem egyenletes az eloszlása, akkor csak egyértelmű hash függvények használatára van lehetőség. Egy egyértelmű hash függvénytől a következőket várjuk el: a gyakorlatban előforduló kulcsokat képezze le a rendelkezésre álló címtartományba; a rendelkezésre álló címtartományon belül tegye egyenletessé az elemek eloszlását. Kulcstranszformációs módszer: a kulcstranszformációs módszer egy algoritmus, amely azt írja le, hogy a hash függvény hogyan képezi le a kulcsértéket a tárbeli címre. A gyakorlatban a kulcsok típusa alapján megkülönböztetünk szöveges és numerikus kulcsokat. Szöveges típusú kulcsok esetén a szöveget alkotó karakterek belső kódjainak valamilyen numerikus függvényét tekintjük, amellyel a kulcstranszformációt a numerikus típusú kulcsok esetére vezethetjük vissza. Numerikus típusú kulcsok esetén többek között az alábbi módszerek használhatók: prímszámmal való osztás, szorzás, helyiérték kiválasztás, bázistranszformáció.  
Szinonimák, szinonimakezelő módszerek: Univerzális hash függvény nem létezik. A csak egyértelmű hash függvényeknél előfordulhat, hogy a függvény a különböző kulcsértékekkel rendelkező adatelemekhez ugyanazt a tárcímet rendeli. Az ilyen adatelemeket szinonimáknak, magát a jelenséget túlcsordulásnak nevezzük. Ezek felbukkanását kezelni kell, mert nem helyezhetünk el két adatelemet ugyanazon a tárhelyen. Néhány módszer: független túlcsordulási lista alkalmazása, nyílt címzés módszere, nyílt címzés (belső láncolással).

**Lista:** Dinamikus, homogén, szekvenciális adatszerkezet. Jelölések: lista – q=[x1,x2…xn], üres lista q=[], a lista feje x1, lista farka [x2…xn], lista vége xn, lista hossza n vagy |q|. Alapműveletek: hozzáférés, elérés, részlistaképzés, konkatenáció, egyesítés, összefűzés. Bővítés bárhol lehetséges, részlistákat képzünk, és azokat összefűzzük. Törlés megvalósítható fizikailag, melynek során részlistákat képzünk és kihagyjuk belőle az elemet, majd összefűzzük. Csere lehetséges részlistákkal, rendezés értelmezhető, keresés értelmezhető, elérés soros vagy közvetlen, bejárás értelmezhető, feldolgozás alapműveletekkel. Vele végezhető speciális műveletek: access head, push, pop (első elem törtlése), access end, inject (bővítés az utolsó elem után), eject (az utolsó elem törlése). Reprezentálható folytonos reprezentációval (vektor), és szétszórtan (egyirányba láncolt lista, kétirányba láncolt lista, cirkulárisan láncolt lista).

**Knuth-Morris-Pratt algoritmus: Dia 8.8**

**Dömölki-féle SHIFT-AND algoritmus: Dia 8.20**

**Fa adatszerkezet:** Homogén, dinamikus, hierarchikus adatszerkezet. Van csúcsa, csomópontjai, a csúcsa általában a gyökérelem, a csomópontok lehetnek közbenső elemek, vagy levélelemek, vannak neki élei, van útvonal, részfa, szint, és magasság. Rendezetlen fáknál nem lényeges ugyanazon csúcsból kiinduló élek sorrendje, rendezett fáknál viszont igen. (Ábra).

**AVL fa adatszerkezet:** A bináris keresőfa olyan rendezett bináris fa, amiben az adatelemek mindegyike rendelkezik egy kulccsal, és minden adatelemre igaz, hogy az adatelem bal oldali részfájában lévő elemek kulcsai kisebbek, a jobb oldali részfájában lévő elemek kulcsai pedig nagyobbak az elem kulcsánál. Az AVL fa egy olyan keresőfa, ami kiegyensúlyozott. Úgy bővítjük, ahogy a keresőfát: mindig levélelemmel. A levélelemmel történő bővítés során a következők fordulnak elő: a fa továbbra is kiegyensúlyozott, ekkor nincs teendőnk, vagy a fa elveszti kiegyensúlyozottságát, amit egy vagy két forgatással megoldhatunk. Négy eset vagy: RR, RL, LR, LL. Törlésnél ugyanúgy törlünk, mint a keresőfából. A törlést követően a következő esetek fordulhatnak elő: a fa továbbra is kiegyensúlyozott, ekkor készen vagyunk. Amennyiben elveszti a kiegyensúlyozottságát, a törlés helyétől a gyökér felé haladva megkeressük az első olyan elemet, amelyhez, mint gyökérelemhez tartozó részfa már nem kiegyesúlyozott, és a bővítésnél ismertetett négy eset közül a megfelelőt alkalmazva kiegyensúlyozzuk a részfát.

**Piros-fekete fa jellemzői:** A piros fekete fa olyan bináris keresőfa, amely a következő tulajdonságokkal rendelkezik: minden csomópontja piros vagy fekete. A gyökere fekete. Minden (NIL) értékű levele fekete. Ha egy csomópont piros, akkor mindkét rákövetkezője fekete. Minden csomópont esetén az összes olyan úton, amely az adott csomópontból indul ki és a levélig vezet, ugyanannyi fekete a csomópontok száma. Egy n adatelemet tartalmazó piros-fekete fa magassága legfeljebb 2log(n+1).

**B-fa adatszerkezet:** A B-fa olyan keresőfa, amely lapokból épül fel. A lapokon mutatók és adatelemek helyezkednek el. Egy-egy lapon az adatelemek maximális számát a B-fa rendje határozza meg. Ha a B-fa rendje n, akkor a gyökérlap kivételével minden lapon legalább n darab adatelemnek kell szerepelnie és minden lapon legfeljebb 2n darab adatelem helyezkedhet el. Egy-egy lapon az adatelemek kulcsaik szerint növekedően rendezettek. Tegyük fel, hogy a B-fa egy lapján az adatelemek aktuális száma éppen m(<=2n). Ekkor a B-fa egy lapja vagy levéllap, vagy pontosan m+1 darab rákövetkezője van. A levéllapon mindegyik mutató értéke NIL. Egy nem levéllap p0 mutatója azt a részfát címzi, amelyben minden kulcsérték kisebb ezen lap k1 értékű kulcsánál. Pi mutatója azt a részfát címzi, amelyben minden kulcsérték nagyobb ezen lap ki-1 értékű kulcsánál és kisebb a ki értékű kulcsánál. Pm mutatója azt a részfát címzi, amelyben minden kulcsérték nagyobb ezen lap km értékű kulcsánál. A B-fában a levéllapok azonos szinten helyezkednek el. A B-fában a K kulcsot keressük. Ha a fa üres, az algoritmus véget ér, a keresett elem nincs a fában. A B fa gyökérlapjain lineáris keresést hajtunk végre. Ha az sikeres, akkor megtaláltuk a keresett elemet, a keresésé véget ér. Ha a lineáris keresés sikertelen, akkor a gyökérlap első olyan eleménél, melynek kulcsa nagyobb, mint a K vagy a gyökérlap utolsó eleme után áll meg a keresés. Előbbi esetben a pi-1 mutató által mutatott részfában, utóbbi esetben a pm mutató által mutatott részfában folytatjuk a keresést.

**Hálós adatszerkezet:** Minden elemnek tetszőleges számú megelőzője és rákövetkezője van. Egy elem lehet egy másiknak (beleértve saját megát is) a megelőzője, rákövetkezője, mindkettő, egyik sem. Ilyen adatszerkezet a gráf, amely a matematikában ismert véges méretű, irányított gráf struktúrájának felel meg. A gráf hálós adatszerkezet. A gráf csúcsok és élek halmaza. Egy él két csúcs közötti kapcsolat. Egy gráfot az határoz meg, hogy mely csúcsai vannak élekkel összekötve. Irányított gráfnál az éleknek irányuk van. Irányítatlan gráfnál az élekhez nincs irány rendelve, vagyis nem teszünk különbséget az A-ból B-be, illetve fordítva menők között. Útnak nevezzük az élek olyan sorozatát, amelyben nem ismétlünk sem élet, sem csúcsot. Körnek nevezzük azt az utat, amelynek kezdő és végpontja azonos. Összefüggő egy gráf, ha (élei esetleges irányításáról megfeledkezve) bármely két csúcs között van út. Egyszerű gráfban bármely két csúcs között legfeljebb egy él lehet, és nem engedjük meg az olyan éleket, amiknek kezdő és végpontja azonos. Súlyozott gráf esetén egy w súlyfüggvény is van. Gráffal végezhető műveletek: létrehozás, bővítés (új elem mellett meg kell adni a megelőzőinek és rákövetkezőinek listáját is), törlés (fizikai), csere megoldható, rendezés nem értelmezett, keresés, elérés, feldolgozás (bejárás alapján), bejárás (szélességi vagy mélységi).  
**Szélességi bejárás:** A bejárás során az adatelemeket fehér, szürke és fekete színűre fogjuk színezni, valamint egy sort fogunk használni a szürke színű adatelemek tárolására. A bejárás során a feldolgozott elemekből feszítőfát építünk. Színezzük a gráf adatelemeit fehér színűre, és hozzuk létre az üres sort. Ha minden adatelem fekete színű, a bejárás véget ér. Ha a sor üres, válaszunk tetszőlegesen egy fehér színű elemet, és színezzük szürkére, helyezzük el a sorban. Ennek az elemnek a feldolgozásakor új feszítőfát fogunk elkezdeni építeni. Ha a sorban van elem, vegyük ki a sor első elemét, fehér színű gyermekeit szürkére festve helyezzük el a sorban, majd az adatelemeket fessük feketére, és helyezzük el a megfelelő feszítőfában. Folytassuk a 2. lépéssel.  
**Mélységi bejárás:** A bejárás során az adatelemeket szürke, fehér és fekete színűre fogjuk színezni és vermet fogunk használni a szürke elemek tárolására. A bejárás során feldolgozott elemekből feszítőfákat építünk. Színezzük a gráf elemeit fehér színűre és hozzunk létre egy üres vermet. Ha minden adatelem fekete színű, a bejárás véget ér. Ha a verem üres, válasszunk tetszőlegesen egy fehér színű elemet, színezzük szürkére, és helyezzük el a veremben. Ennek az elemnek a feldolgozásakor új feszítőfát fogunk elkezdeni építeni. Ha a veremben van elem, vegyük ki a verem első elemét, fehér színű gyermekeit szürkére festve helyezzük el a veremben, majd az adatelemet fessük feketére, és helyezzük el a megfelelő feszítőfában. Folytassuk a 2. lépéssel.

**Dynamic Multithreaded Programming**

Szálak: virtuális processzorok szoftveres absztrakciója. A szálak közösen használják ugyanazt a memóriát. Saját vermük és programszámlálójuk van. Egymástól függetlenül hajtják végre a feladataikat. Az operációs rendszer tölti be a szálakat a processzorokba, hajtatja végre a feladatokat, vált más szálakra, ha szükséges. A dinamikus többszálú programozási modell (DMP) megengedi a programozónak, hogy leírja a logikai párhuzamosságot és ne törődjön a kommunikációs protokollok vagy a betöltési ˝ egyensúly problémáival. A következőkben azt fogjuk tanulmányozni, hogyan írhatunk többszálú algoritmusokat a modell keretein belül és hogyan segítik a hatékony számolásokat a háttérben meghúzódó rendszer szintű támogatások

**Párhuzamos mátrixok szorzása: Dia 12.**

**Párhuzamos összefésülő rendezés: Dia 12.**

**Kis kérdések**

**RAM modell:** A RAM (random access machine) modell olyan utasításokat tartalmaz, amelyek rendszerint megtalálhatók a valódi számítógépeken:

* aritmetikai (összeadás, kivonás, szorzás, osztás, maradékképzés, alsó egész rész, felső egész rész)
* adatmozgató (betöltés, tárolás, másolás)
* és vezérlésátadó (feltételes és feltétel nélküli elágazás, eljáráshívás és visszatérés) utasítások.

Minden ilyen utasítás konstans hosszúságú időt vesz igénybe. A RAM modellben az utasítások egymás után hajtódnak végre, egyidejű műveletek nélkül.

**Jelölések:**

* **θ jelölés**: Jelentése aszimptotikus éles korlát. Egy adott g(n) függvény esetén θ(g(n))-nel jelöljük a függvényeknek azt a halmazát, amelyre θ(g(n))=f(n) : létezik c1, c2, és n0 pozitív állandó, hogy 0≤c1\*g(n)≤f(n)≤c2\*g(n) teljesül minden n≥n0 esetén. T(n)= θ(n2)↔T(n) € θ(n2)
* **O jelölés**: Jelentése aszimptotikus felső korlát. Egy adott g(n) függvény esetén O(g(n))-nel (nagy ordó g(n)) jelöljük a függvényeknek azt a halmazát, amelyre O(g(n))= f(n): létezik c és n0 pozitív állandó, hogy 0≤f(n)≤c\*g(n) teljesül minden n≥n0 esetén.
* **Ω jelölés**: Jelentése aszimptotikus alsó korlát. Egy adott g(n) függvény esetén Ω(g(n))-nel jelöljük a függvényeknek azt a halmazát, amelyre Ω(g(n))=f(n): létezik c és n0 pozitív állandó, hogy 0≤c\*g(n)≤f(n) teljesül minden n≥n0 esetén.

**Éles, nem éles:** Az O jelölés által adott felső korlát aszimptotikusan vagy éles (2n2=O(n2)) vagy nem (2n=O(n2)). Amikor hangsúlyozni szeretnénk, hogy az adott felső korlát nem éles, akkor az o jelölést használjuk („kis ordó”): o(g(n))= f(n): tetszőleges c>0-hoz található n0>0, hogy 0≤f(n)<c\*g(n) teljesül minden n≥n0 esetén. Vagy (nemnegatív aszimptotikus viselkedés mellett limesz f(n)/g(n)=0.

**Leszámláló rendezés:** Feltételezzük, hogy az n bemeneti elem mindegyik 0 és k közötti egész szám, ahol k egy ismert egész. Ha k=O(n), a rendezés futási ideje θ(n).  
procedure Leszámláló\_rendezés (A,B,k)  
1. for i←0 to k do

2. C[i] ←0

3. end for

4. for j←1 to méret(A) do

5. C[A[j]] ←C[A[j]]+1

6. end for

7. for i←1 to k do

8. C[i] ←C[i]+C[i+1]

9. end for

10. for j←méret(A) downto 1 do

11. B[C[A[j]]] ←A[j]

12. C[A[j]] ←C[A[j]]-1

13. end for

end procedure

**Számjegyes rendezés:** Feltételezi, hogy az n elemű A tömb minden egyes eleme d jegyű, ahol az első számjegy a legalacsonyabb helyértékű számjegy és a d-edik számjegy a legmagasabb helyértékű számjegy.

procedure Számjegyes\_rendezés (A,d)  
1. for i←to d do

2. stabil algoritmussal rendezzük az A tömböt az i-edik számjegy szerint

3. end for

end procedure

**Edényrendezés:** Várható futási ideje lineáris, amennyiben a bemenet egyenletes eloszlásból származik. Azt feltételezi, hogy a bemenetet egy olyan véletlen folyamat generálja, amelyik egyenletesen osztja el az elemeket [0, 1] intervallumon.

procedure Edény\_rendezés (A)  
1. n←méret(A)

2. for i←1 to n do

3. szúrjuk be az A[i] elemet a B[ elemet a B[└nA[i]┘] listába

4. end for

5. for i←0 to n-1 do

6. rendezzük a B[i] listát beszúrásos rendezéssel

7. end for

8. sorban összefűzzük a B[0], B[1], …, B[n-1] listákat

end procedure

**Kiegyensúlyozott bináris fa:** Minden szintjén az egyes részfák magassága nem ingadozik többet egy szintnél.

**Tökéletesen kiegyensúlyozott fa:** A fa bármely elemének bal és jobboldali részfájában az elemek száma legfeljebb eggyel tér el.

**Szigorúan vett bináris fa:** Bármely elemének vagy 0, vagy 2 rákövetkezője van.

**Minimális magasságú fa:** Olyan fa, amelynek a magassága az adott elemszám esetén a lehető legkisebb.

**Táblázatban figyelni:** Soros táblázat esetén a beszúrásnál az új elemet az eddigi utolsó elem mögé helyezzük el, cserénél az érték rész bármikor cserélhető, a kulcsérték csak akkor, ha az új érték nem szerepel. Önátrendező táblázatnál ugyanez történik. Rendezett táblázatnál a bővítésnél az érkező elemeket rendezett sorozatba történő beszúrással helyezzük el, tehát a legelejére kerülnek a beszúrt elemek, cserénél az adatelem értékrésze bármikor cserélhető.

**Speciális halmaz műveletek:** A halmaz adatszerkezetek alapműveletei az eleme, Є, ami megmondja, hogy az adatelem benne van-e a halmazban vagy sem. Az unió U, két halmaz unióját adja, a metszet ∩, két halmaz metszetét adja, a különbség \, két halmaz különbségét adja. x Є AUB = x Є A V x Є B. x Є A∩B = x Є A ˄ x Є B. x Є A\B = x Є A ˄ x !Є B.

**Önátrendező táblázat:** A műveletek többsége megegyezik a soros táblázatéval, beszúrás ??, feldolgozás után az elemet a táblázat elejére helyezzük, az addigi első elem elé. Ennek következtében a legutoljára feldolgozott elem lesz a táblázat elején. Az elemek sorrendje ezután a táblázatban jól fogja közelíteni a feldolgozási gyakoriságot. Legjobban szétszórtan, egyirányba láncolt listával reprezentálható. Ekkor a feldolgozás művelete mindössze három mutató értékének a cseréjét jelenti.

**Verem:** Speciális lista adatszerkezet. Csak a verem tetejére lehet betenni, illetve csak onnan lehet kivenni. Az utoljára betett elem a verem tetejére kerül, az elsőnek betett a verem aljára. LIFO. Létrehozásnál üres verem inicializálása, bővítés PUSH-sal, utoljára betett elem fölé, csere nincs, törlés fizikai, verem tetején lévő elemet (POP), rendezés, keresés, bejárás nem értelmezett, elérés TOP (legfelső elem), feldolgozás LIFO, utolsóként érkező elemet dolgozzuk fel.

**Sor:** Speciális lista adatszerkezet, melynek alapműveletei a speciális listaműveletek közül a következők: első elemhez történő hozzáférés (ACCESS HEAD), bővítés az utolsó elem mögé (PUT – INJECT), az első elem törlése (GET – POP). Az első elemhez történő hozzáfélés és az első elem törlésének műveletét gyakran egy műveletként definiáljuk. Létrehozás üres sorral, bővítés az utolsó elem mögé, csere nincs, törlés fizikai (első elemet), rendezés, keresés, bejárás nem értelmezett, elérés csak az első elemnél, feldolgozás FIFO.

**Fa magassága, elemek száma:** Azt mondjuk, hogy egy fa kiegyensúlyozott akkor, ha bármely elemére igaz, hogy az elem bal oldali és jobb oldali részfájának magasságkülönbsége 1. Így annyi eleme kel hogy legyen, hogy ez az egyensúly ne boruljon fel. Egy n adatelemet tartalmazó piros-fekete fa magasság legfeljebb 2log(n+1).

**Fa magassága, elemek száma:** Azt mondjuk, hogy egy fa szigorúan bináris akkor, ha minden adatelemnek vagy 0 vagy 2 rákövetkezője van. Így mélysége meghatározható úgy, hogy az elemszámát megnézzük, 2 melyik hatványába fér bele, és a hatványa+1 a mélysége. Tökéletesen kiegyensúlyozott esetben a jobb oldali és a bal oldali részfák elemszáma között a különbség 1 lehet.

**Háromszögmátrixok tárolási módja:** Két fajta háromszögmátrixot különböztetünk meg, felső és alsóháromszög mátrixot, ezek mindketten négyzetes (kvadratikus) mátrixok. Ezek olyan négyzetes mátrixok, amiknek a főátlója felett vagy alatt csak 0 elem található. Az értékes elemeket sor agy oszlopfolytonosan egy n\*(n+1)/2 elemű V vektorra szoktuk leképezni. A felső h.mátrixot oszlopfolytonosan célszerű leképezni, az alsót sorfolytonosan.

**Sztring adatszerkezet:** A sztring egy olyan szekvenciális lista, amelynek az elemei egy ábécé szimbólumai. Ezeket a szimbólumokat karaktereknek nevezzük. Vele végzett műveletek: létrehozás, bővítés (bárhol), törlés (fizikai), csere (törlés+bővítés), keresés (karakter vagy részsztring), elérés (soros vagy közvetlen), bejárás (értelmezhető).

**Sztring ábrázolása:** Ábrázolhatjuk fix hosszon, változó hosszon, ilyenkor hossz ott van a string előtt, vagy információs táblázattal esetleg végjellel. (Dia 8. 4.)

**Sztringek ábrázolás hátrány:** Szétszórt reprezentációnál listát használunk és a listaelemek tartalmazhatnak egy karaktert, de ezzel rossz a helykihasználás, vagy tartalmazhatnak részsztringet. Ebben az esetben viszont a részsztringek eltérő hosszúságúak lehetnek, és valamelyik folytonos reprezentációval ábrázoljuk azokat, így azt mindenképp használni kell.

**Három mintaillesztő algoritmus:** Dia 8. 7.,

**Karakterisztikus függvények:**

**AVL-fa:** A bináris keresőfa olyan rendezett bináris fa, amiben az adatelemek mindegyike rendelkezik egy kulccsal, és minden adatelemre igaz, hogy az adatelem bal oldali részfájában lévő elemek kulcsai kisebbek, a jobb oldali részfájában lévő elemek kulcsai pedig nagyobbak az elem kulcsánál. Az AVL fa egy olyan keresőfa, ami kiegyensúlyozott.

**AVL-fába beszúrás algoritmusa:** Dia 6. 34

**Piros-fekete-fába beszúrás:** Egy piros-fekete fát úgy bővítünk, mint egy keresőfát: mindig levélelemmel, amelyet pirosra színezünk. A levélelemmel történő bővítést a követően a következő esetek fordulhatnak elő: a fa továbbra is rendelkezik a piros fekete tulajdonsággal, nincs teendő. Második eset: nem teljesül az, miszerint a gyökérelem fekete. Ez akkor fordulhat elő, ha gyökeret szúrunk be. Ekkor átszínezzük a beszúrt elemet feketére. A harmadik esetben nem teljesül az, miszerint nincs a fában két egymást követő piros csomópont. Ez csak akkor fordulhat elő, ha a beszúrt elem szülője piros. Mivel a gyökér fekete, a beszúrt elemnek biztosan van nagyszülője, amelynek feketének kell lennie. Ekkor forgatásokat és átszínezéseket kell végrehajtanunk. Ez az Okasaki módszer.

**Rekord:**

**Bináris keresésre feltételek:** Az adatoknak, ami között keresnünk kell, rendezettnek kell lenniük növekvő sorrendben. Folytonos tárolásnál hatékony kereső algoritmus.

**Kupac adatszerkezet:** A kupac olyan fa, amely rendelkezik a kupac tulajdonsággal: a gyökérelemet kivéve bármely adatelemének a kulcsa kisebb vagy egyenlő az adatelem szülőjének a kulcsánál. Az ilyen fában a legnagyobb kulcsó elem mindig a gyökérelem, ezért max-kupacnak is nevezzük. Ha megfordítjuk a relációt, akkor a gyökérelem lesz a legkisebb kulcsú elem, ekkor min kupacot kapunk.

**Bináris kupac:** A bináris kupac olyan bin. fa, amely a kupac tulajdonságon kívül az alak tulajdonsággal is rendelkezik: a fa teljes bináris fa, azaz minimális magasságú és ha a legalsó szint nincs teljesen kitöltve, akkor azon a szinten a csomópontok balról jobbra kerülnek feltöltésre.

**Önátrendező táblázat:** Az önátrendező táblázatnál a műveletek többsége megegyezik a soros tábláéval, van létrehozás, bővítés (táblázat elején), törlés (kiesik az elem), csere, keresés, elérés, bejárás, és feldolgozás, ekkor pedig a feldolgozott elem a táblázat elejére kerül.

**Lista alap műveletei:** A lista adatszerkezetnél a hozzáférés, elérés az közvetlen q[i]=xi. Részlistaképzés, allistaképzés: q[i…j]=[xi,xi+1…xj-1,xj]. Konkatenáció, egyesítés: r=[y1…ym], q&r=[x1,x2,…xn,y1…ym]. Bárhol bővíthető részlisták képzésével, azok összefűzésével, törlés fizikai ugyanígy, csere ugyanígy, rendezés értelmezhető, keresés értelmezhető, elérés soros vagy közvetlen, bejárás van, feldolgozás alapműveletekkel.

**Sor adatszerkezet folytonos ábrázolása:** Dia 4. 28.

**Előnye soroknál szétszórt ábrázolásnak:** Hátrány az egyértelműen, hogy elérni és feldolgozni az első mutató által hivatkozott elemet tudjuk csak, bővíteni pedig az utolsó mutató által hivatkozott elem mögé tudunk.

**Spawn utasítás:** Párhuzamos programozás során használt utasítás, amely segítségével ivadékot, leszármazott processzust tudunk létrehozni.

**Paralell utasítás:** Párhuzamos programozás során használt utasítás, amely segítségével párhuzamosítani tudjuk a programok processzusait.

**Mohó ütemezés:** Akkor beszélünk mohó ütemezésről, amikor az ütemező annyi processzort rendel hozzá a szálakhoz minden időpillanatban, amennyi éppen elérhető. (Nem tervez semmilyen szempontból.) Ha egy adott időlépésben legalább P szál van előkészítve a futásra, akkor azt mondjuk, hogy az adott időlépés teljes (complete step). Ilyenkor a mohó ütemező kiosztja az összes processzort valamelyik P darab szálnak az előkészítettek közül. Ha adott időben P-nél kevesebb szál van előkészítve, akkor nem teljes időlépésről beszélünk (incomplete step) és a mohó ütemező minden strand-hoz hozzárendel egy saját processzort.  
Mohó ütemezés esetén belátható hogy a munka ill. a feszítés törvény alapján a futási idő: TP<=T1/P+Tvégtelen.

**Versenyhelyzet:** (Rare condition) Egy többszálú algoritmust determinisztikusnak (deterministic) nevezünk, ha az ütemező nem tudja befolyásolni a végeredményt. Nem-determinisztikus akkor, ha a dag-ból kiolvasható szabályok megtartása melletti lehetséges eltérő utasítás végrehajtási sorrendek némelyike más eredményre vezet. Det.-nek tervezett algoritmusok gyakran válnak nem-det.-sá, mert ún. versenyhelyzet alakul ki bennük. Ennek lényege: logikailag párhuzamos utasítások ugyanazt a memória területet érik el, és közülük legalább egy ír is oda.

**Párhuzamos összefésülő algoritmus: Dia 12. 41**

**Bather-féle páros-páratlan algoritmus: Dia 12. 51.**