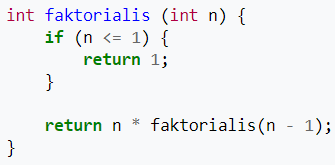
**Programozás, algoritmusok, adatstruktúrák - licensz vizsga 2021**

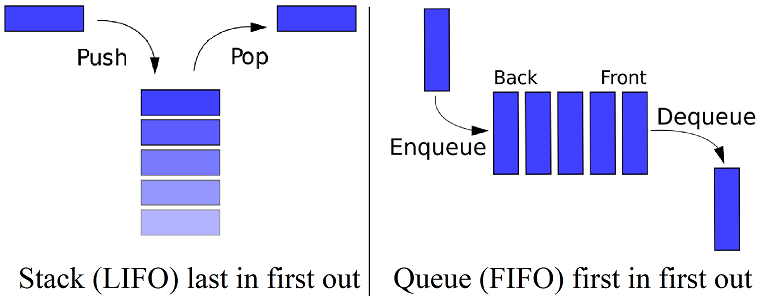
**1. Rekurzió.**

* A rekurzió egy olyan művelet, amely végrehajtásakor a saját maga által deffiniált műveletet, vagy műveletsort hajtja végre, ezáltal önmagát ismétli.
* Leggyakrabban a backtrack (visszalépéses keresés által kapjuk meg a legrövidebb utat) típusú algoritmusok esetében szokták alkalmazni.
* A ciklusok rekurzióvá alakíthatók, illetve a rekurziók ciklusokká.
* Ahogy ciklusok esetében, a rekurziónál is előkerülhet a végtelenség fogalma, ez akkor történik meg, ha túl sokszor fordul elő a meghívás, ez a verem túlcsordulásához vezet.



**2. Vermek, sorok**

* **A verem** egy LIFO típusú struktúra, amely véges számú és azonos típusú adatot tárol.
* A verembe értelmezett műveletek:
  + **Push:** A verem tetejére helyez egy új adatot. Ha a verem betelt, akkor túlcsordulásos állapotba kerül.
  + **Pop:** A verem legfelső elemét leveszi és visszaadja. Ha a verem már üres, akkor alulcsordulás állapotba kerül.
  + **isEmpty,**
  + **isFull.**
* A veremmutató mindig a legfelső elemre mutat.
* A fizikailag a memóriában helyezkedik el
* A stack implementálásához egy speciális regiszterben elhelyezkedő mutató szükséges, mely mutatót (a regiszter tartalmát) speciálisan a stacket kezelő utasítások mozgatják, változtatják, szükség szerint.
* **A sor** az egy FIFO típusú struktúra, ahol mindig a legelsőként behelyezett elemet vesszük ki először.



**3. Láncolt listák, körkörösen láncolt listák, fák.**

A **láncolt lista** a leggyakrabban használt adatszerkezet, cellák sorozatából épül fel, amelyek tetszőleges számú adatmezőt és 1, vagy 2 hivatkozást tárolnak. A hivatkozások a lista következő, illetve előző sorára mutatnak.

A láncolt szerkezet lehetővé teszi tetszőleges helyre elemek beszúrását, több változata fellelhető: **egyszeresen láncolt, kétszeresen láncolt, körkörösen láncolt lista.**



*Kétszeresen láncolt lista pl.*

**A körkörösen láncolt lista** az egyszerűbb listák (egyszeresen és kétszeresen láncolt) kibővítése. Az eltérése abban nyilvánul meg, hogy az első, illetve az utolsó elem össze van csatolva egymással. A bejárás bármelyik elemnél kezdhetjük (előny), azonban vigyáznunk kell a körbeérésre (hátrány).



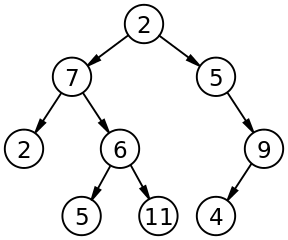
*Körkörösen láncolt lista pl.*

A **fák** azok körmentes összefüggő gráfok, jellemző rájuk a hierarchikus (szülő-gyermek) elrendezés áll.

* A **gyökérelem** a fa azon eleme, amelynek nincs őse.
* A **levél** a fa azon eleme, melynek nincs utódja
* **A közbenső elemek**  a fa nem gyökér-illetve levélelemei
* Az **út** a gyökérből kiinduló bejárás.

**4. Bináris fák és bináris keresőfák.**

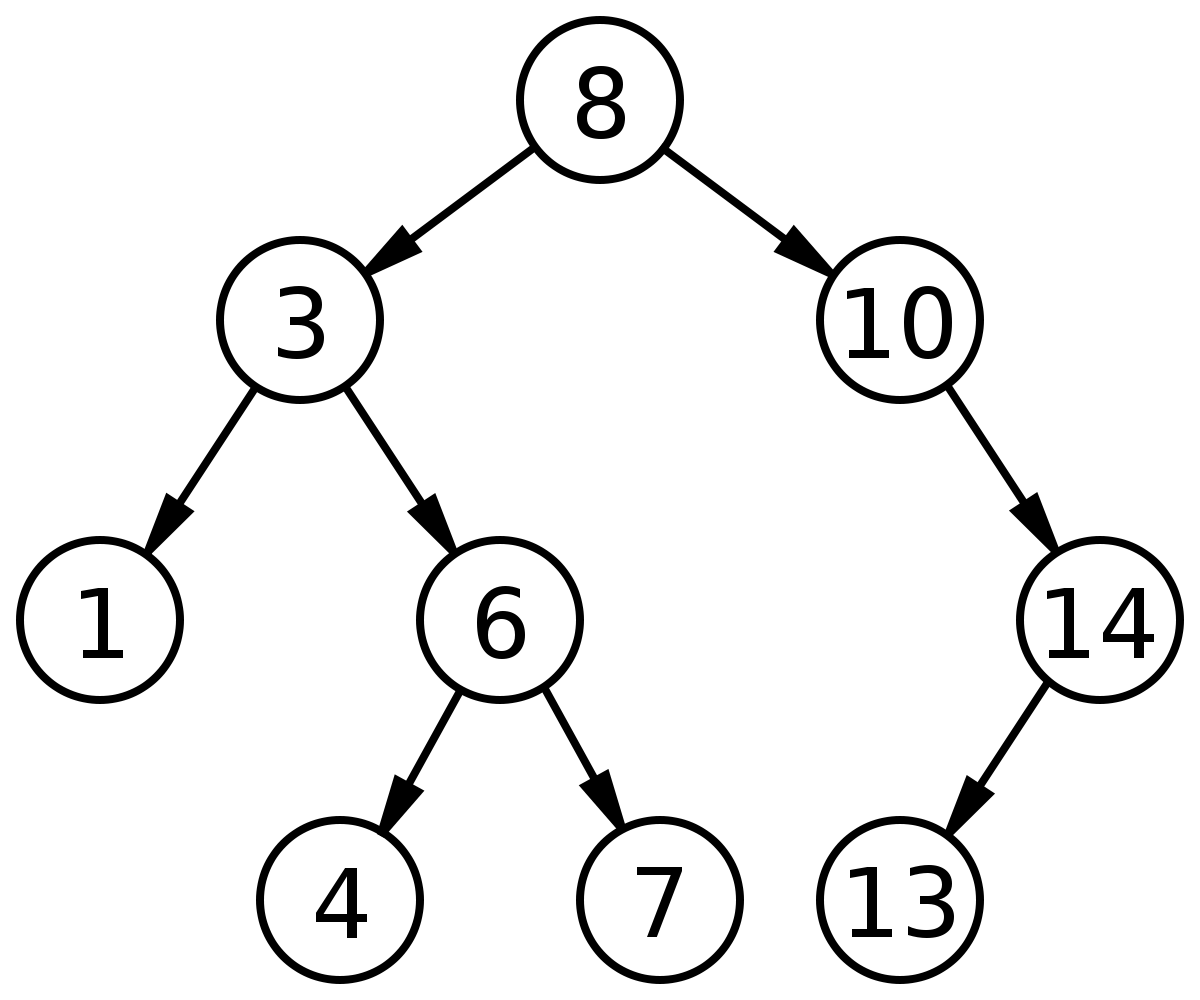
Bármely típusú fa ábrázolható **bináris fa** segítségével. A bináris fa legfőbb jellemzője az, hogy bármelyik csomópontnak csak két utóda lehet. A bináris fák utódjait megkülönböztetjük aszerint, hogy bal illetve jobb részfák.



A bináris fákat ábrázolhatunk **tömbök segítségével** is, ebben az esetben meg kell adnunk minden csomópontra, az indexét, az információját és a bal illetve a jobb utódját. Például a képen látható bináris fa esetében, tömbökkel való ábrázolása mellette.

Egy speciális esete a **kiegyensúlyozott bináris fa**, mely esetén minden csúcsnak pontosan 0 vagy 2 gyereke lehet, a magasság különbség legfeljebb 1.

**A keresőfa** olyan fa, amelynek bármely elemére igaz, hogy az elem bal oldali részfájában az összes kulcs kisebb, a jobboldali részfájában az összes elem kulcsa nagyobb mint az adott csúcs kulcsa.



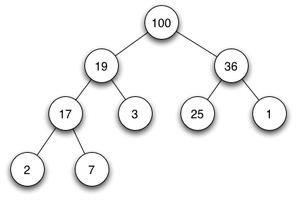
**5. Kupacok, binomiális fák.**

A **kupac** egy speciális fa alapú adatszerkezet, mely eleget tesz a kupac tulajdonságnak, azaz ha **B** csúcs fia az **A** csúcsnak, akkor kulcs(A) >= kulcs(B), (tehát az apa az nagyobb a fiánál) ebben az esetben a kupacot *max-kupacnak* nevezzük, ha az egyenlőséget megfordítjuk, akkor *min-kupacnak*.

A **majdnem bináris kupacok** azok azt jelentik, hogy csak a legalsó szinten hiányozhatnak elemek.

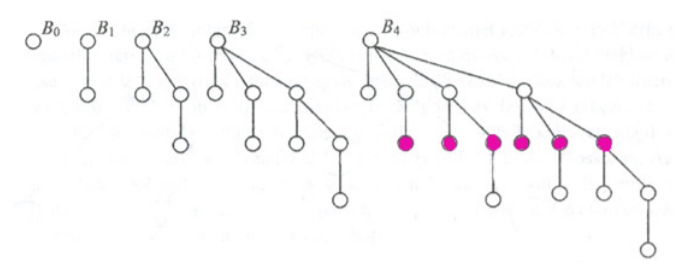
A kupac adatszerkezet különböző fajtáit több algoritus hatékony implementációja során alkalmazhatjuk:

* heap sort - kupac rendezés - az egyik leghatékonyabb keresési algoritmus
* súlyozott gráfok bejárási algoritmusának gyorsítása



A **binomiális fát** rekurzív deffinícióval adhatunk meg:

* A B0 fa egyetlen csomópontot tartalmaz.
* A Bk (k>0) pedig két összekapcsolt Bk-1fából áll, ahol a fa gyökércsúcsa a másik fa gyökércsúcsának legbaloldalibb gyereke lesz, tehát folyamatosan úgymond nyomtatva lesz az előző fa



**6. Rendezési algoritmusok.**

A **rendezési algoritmusok** azok olyan algoritmusok, amelyek valamilyen szempont alapján sorba állítják az elemek egy listáját, abból a célból, hogy több és hatékonyabb műveleteket tudjunk elvégezni. Ezek sokkal inkább elvégezhetők egy rendezett adathalmazzal, mint egy rendezetlennel.

Mekülönböztetünk **külső** és **belső** rendezéseket:

* A belső esetén a rendezés az elemek összehasonlítására alapszik
* A külső rendezés esetén az algoritmusok nem használják az elemek között fennálló hasonlítást

**Belső rendezések:**

* *Bubble sort*: A buborékrendezés népszerű rendező algoritmus. Úgy dolgozik, hogy egymás után felcseréli azokat a szomszédos elemeket, amelyek rossz sorrendben vannak.
* *Selection sort (kivalaszto)*: Kiválasztjuk az elejéről kezdve a sorozat egyes elemeit, majd egyenként összehasonlítjuk a sorozat még nem rendezett többi elemével. Ha a sorrend nem megfelelő, akkor cserélünk.
* *Insertion sort (beszurasos)* A beszúrásos rendezés abból áll, hogy a tömb elejétől vesszük a számokat és hasonlítgatjuk a következő elemekkel, és tologatjuk a listában egyenként, mindig az éppen aktuális előtte levő elemhez hasonlítjuk, ha ott megfelel a feltétel, akkor marad.
* *Merge sort (osszefesulo):* rekurzív összehasonlító algoritmus, ami abból áll, hogy ketté választja a listát és külön külön rendezi, a kapott részeket meg összefésüli egy rendezett listává, amely a harmadik lista lesz
* *Heapsort (kupacrendezes)*: egy kupacból indul ki és a feltétel szerint cseréli fel a csomópontokat, lényegében max-kupacot kell készíteni úgy, hogy a leveleket folyamatosan felcseréljük az aktuális gyökérrel, ekkora a gyökeret töröljük és megtalálja a helyét a listában. Mindig balról indulunk.
* *Quicksort (gyorsrendezes)*: A gyorsrendezés oszd meg és uralkodj elven működik: a rendezendő számok listáját két részre bontja, majd ezeket a részeket rekurzívan, gyorsrendezéssel rendezi. A felbontáshoz kiválaszt egy támpontnak nevezett elemet (más néven pivot, főelem vagy vezérelem), és particionálja a listát: a támpontnál kisebb elemeket eléje, a nagyobbakat mögéje mozgatja

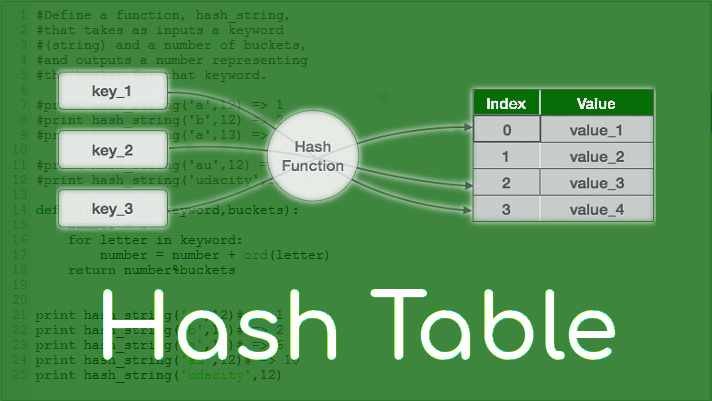
**7. Hasító (hash) táblák és hasító algoritmusok.**

A **hash-table** egy olyan adatszerkezet, amely egy hash-függvény segítségével rendel minden értékhez egy kulcsot, mellyel később (ideális esetben) konstans időben azonosítja az értéket, ezáltal implementálva egy asszociatív tömböt.

* Egy leképzést végez két halmaz között
* Ideális esetben minden kulcs egyedi, valószágban történhetnek ütközések,így számolni kell ezen ütközésekkel is.
* Sok esetben hatékonyabbak, mint a keresőfák, ezért gyakran használják asszociatív tömbök impementációjában, adatbázisok indexelésében, illetve cache memória felépítésében.

A **hasítófüggvény** egy speciális eljárás hasítótáblák felépítésére.

* Nagy méretű adatállományok adatelemeinek gyors, hatékony megkeresésére használják
* adott K érték esetén kiszámolunk egy f(K) függvényt, amely meghatározza K érték helyét az adatállományban
* szükséges továbbá az azonos hasítóértékekkel rendelkező elemek megkülönböztetése



**8. Algoritmusok bonyolultsága.**

**Bonyolultság**:

* Időbonyolultság: Mennyi alapművelet szükséges az algoritmusok elvégzéséhez (pl összeadás, rotate left stb)
* Tárbonyolultság: Memóriaigény, tárhely a fájlmüveleteknél stb.

Ennek egyszerűbbé tétele érdekében a **bonyolultság nagyjából megközelíti az algoritmus végrehajtásához szükséges lépések számát.**

* A komplexitást nagy O-val jelöljük.
* A komplexitás tipusai (futási idő):

|  |  |  |
| --- | --- | --- |
| Konstans | O(1) | Egy adott művelet végrehajtásához hány lépést kell megadnunk, és ez a szám nem függ a bemeneti adatok méretétől. |
| Logaritmikus | O(log(N)) | Általában 2-es alapú logaritmusokat használnak N elemen. Ilyen esetben 1 millió N esetében a egy O komplexitású logaritmus kb 20 lépést végez. |
| Lineáris | O(N) | Az N elemen végzett művelet végrehajtásához közel ugyanannyi lépés szükséges, mint az elemek száma. Például, ha 1000 elemünk van, akkor körülbelül 1000 lépést kell megtenni. A lineáris komplexitás azt jelenti, hogy az elemek **száma és a lépések száma lineárisan függenek**, például az N elem lépéseinek száma lehet N / 2 vagy 3 \* N. |
| Polinomiális | O(n^2) v O(n^3) | Valamilyen polinom szerinti értéket vesz figyelembe, tehát ha másodfokú a polinom és N = 10, akkor 100 lépés szükséges. N a bemeneti adatok nagysága. |
| Exponenciális | O(2n), O(N!), O(nk), … | Exponenciálisan nő az érték. |

Lineáris (x=y függvény szerint változik), polinomiális (valamilyen polinom függvény, elsőfokú v másodfokú s úgy tovább), exponenciális (exponenciális az valaminek a hatványa, pl 2^x, pl ahogy nő a paraméter lineárisan (1,2,3) úgy nő az algoritmus ideje 2x ekkel), logaritmilkus(a logaritmikus pedig az elején gyorsan nő, aztán lelassul) függvényben változó bonyolultságok úgy idő mint tárhely esetében fellelhetők.

-• polinomiális feladat (P) és nemdeterminisztikusan polinomiális feladat (N P)

-P: polinom időben v tárhellyel meg tudunk oldani.

NP: van olyan tanú, aminek a segítségével polinom id®ben le tudjuk ellen®rizni, hogy a válasz igenl®.

co-NP: van olyan tanú, aminek a s egítségével polinom id®ben le tudjuk ellen®rizni, hogy a válasz nemleges.

NP-nehéz: minden NP-beli probléma polinom id®ben visszavezethet®.

NP-teljes: NP-beliek és NP-nehezek.

annyi a lényeg hogy azoknál nem lehet meghatározni hogy polinomiális-e ezért àltalában exponenciális vagy hasonló megoldásaik vannak

N P-teljes feladatok: • logikai formulák kielégíthetosége ˝ • hátizsák probléma • csomagolási probléma (bin packing) • Hamilton-út keresése • lefedoszó probléma