Írta:
ADONYI RÓBERT

ADATSTRUKTÚRÁK ÉS ALGORITMUSOK

Egyetemi tananyag

2011

LEKTORÁLTA: Dr. Fábián Csaba, Kecskeméti Főiskola Gépipari és Automatizálási Műszaki Főiskolai Kar Informatika Szakcsoport

TÁMOGATÁS:
Készült a TÁMOP-4.1.2-08/1/A-2009-0008 számú, „Tananyagfejlesztés mérnök informatikus, programtervező informatikus és gazdaságinformatikus képzésekhez” című projekt keretében.

ISBN 978-963-279-488-4

KÉSZÜLT: a Typotex Kiadó gondozásában

FELELŐS VEZETŐ: Votisky Zsuzsa

AZ ELEKTRONIKUS KIADÁST ELŐKÉSZÍTETTE: Erő Zsuzsa

KULCSSZAVAK:

adatstruktúra, algoritmus, C, C++, programozás, verem, sor, láncolt lista, gráf, fa.

ÖSSZEFOGLALÁS:

A jegyzet célja, hogy betekintést nyújtsunk a programozási feladatok során elénk kerülő programtervezési kérdésekbe. Ehhez bemutatjuk az alap adatszerkezeteket és az adatszerkezeteken megvalósítható különböző algoritmusokat, hiszen a rendelkezésünkre álló adatszerkezetek és a hozzájuk kapcsolódó algoritmusok tulajdonságainak és működésüknek alapos ismeretében tudjuk csak az alkalmazás igényeinek legmegfelelőbb megoldást kiválasztani. A jegyzet az elméleti ismeretanyagot és a programozás jellegű tárgyak teljesítéséhez szükséges gyakorlati jellegű tudást kapcsolja össze. A jegyzetben szereplő mintaprogramok és kódrészletek objektum orientált szemléletben C++ programozási nyelven kerülnek megvalósításra.

Tartalomjegyzék

Bevezetés

A jegyzet célja, hogy Az adatstruktúrák és algoritmusok tárgy alapjait képező ismeretanyagról egy összefoglaló képet adjon. Az adatstruktúrák és algoritmusok elengedhetetlen alapjai a szoftvertervezés, szoftverfejlesztés folyamatának, komplex informatikai programoknak, szoftvereknek. A jegyzet összekapcsolja a témához kapcsolódó elméleti ismeretanyagot és a programozás jellegű tárgyak teljesítéséhez szükséges gyakorlati jellegű tudást. A megfelelő programozási ismeretekhez elengedhetetlen, hogy tudjuk és ismerjük

- hogyan lehet az információt a számítógép memóriájában tárolni,
- milyen adatszerkezetekbe lehet az adatot szervezni,
- mik az előnyei és hátrányai a kiválasztott adatszerkezetnek,
- hogyan lehet algoritmusokat felépíteni és az algoritmusokkal a tárolt adatot módosítani,
- mik a kiválasztott adatszerkezeteknek és a hozzájuk kapcsolódó algoritmusok számítási igénye.

Az alap adatszerkezetek és a hozzájuk kapcsolódó legfontosabb algoritmusok tulajdonságainak és működésüknek alapos ismeretében tudjuk csak az alkalmazás igényeinek legmegfelelőbb megoldást kiválasztani. A jegyzetben szereplő mintaprogramok és kódrészletek objektum orientált szemléletben C++ programozási nyelven kerülnek megvalósításra.

Előfeltételek

A jegyzet megértéséhez feltételezzük, hogy az olvasó ismeri a C++ programozási nyelv alapjait, hogyan lehet egy C++ programot létrehozni. Feltételezzük, hogy az olvasó ismeri a C++ szintaktikai szabályait, milyen változókat lehet használni, milyen paraméter átadási lehetőségeink vannak, hogy működnek a mutatók (*pointer*), hogyan lehet osztályokat és objektumokat létrehozni, milyen öröklődési szabályok vannak.

Programozási konvenciók

Egy jó szoftverfejlesztési környezet segíti automatikus kódformázással, színekkel a programkód olvashatóságát. Azonban a környezet nyújtotta támogatás mellett érdemes néhány kódolási konvenciót betartani, hogy a programkód a későbbiek során is érthető, értelmezhető legyen.

A javaslatok elsődleges célja az, hogy a programkód olvasható legyen, minősége javuljon. Javaslatok közül néhány:

- adattípusok elnevezésére kis és nagybetűket is használhatunk, az első betű nagy legyen (pl. UserAccount)
- változók elnevezésére kis és nagybetűket és használhatunk, azonban az első betű kicsi legyen (pl. userAccount)

- konstansok elnevezésére nagybetűket használjunk, ha több szót tartalmaz az elnevezés, akkor a szavak elválasztására az aláhúzás karaktert használjuk (pl. ACCOUNT_COUNT)
- függvények, eljárások elnevezése fejezze ki a cselekvést, használjunk igéket, kis és nagybetűket is tartalmazhatnak, azonban kis betűvel kezdődjenek (pl. getAccount())
- a névterek (namespace) elnevezésére csak kisbetűket használjunk (pl. model::geometry)
- sablon (template) elnevezésére egy nagy betűt használjunk (pl. template <class C>)
- ha egy névnek része egy rövidítés, akkor a rövidítést ne nagybetűkkel használjuk (pl. importHtml())
- a kód komplexitását csökkenthetjük, ha a változó neve megegyezik a típusával (pl. Database database)
- használjuk a get/set/compute/find/initialize … szavakat a megfelelő funkció kifejezésére
- ciklusváltozóknak használjuk az i, j, k, l változókat

A fenti felsorolás csak néhány kiragadott ajánlás kódolási konvencióra. Találhatunk ajánlást a forrásfájl felépítésére, tagolására, a vezérlési szerkezetek szervezésére.

Algoritmus és adat

Az algoritmusok fontos részei az informatikának és mindennapi életünknek is. Nem csak a számítástechnikában, hanem napi rutinjaink elvégzésében is algoritmusokat hajtunk végre. Az algoritmus nem más, mint jól meghatározott lépések sorozata egy bizonyos cél elérésének a céljából. Például, ha egy süteményt sütünk a receptben szereplő lépéseket hajtjuk végre, ebben az esetben a recept írja le az algoritmust.

Többfajta lehetőségünk van algoritmus leírására. A legegyszerűbb, ha egy beszélt nyelvet (magyar, angol) használunk arra, hogy elmagyarázzuk hogyan működik az algoritmus. A számítástechnikában azonban inkább valamilyen matematikailag precízebb leírást kell választanunk, például valamely programozási nyelvet. A programozási nyelvvel felépített algoritmust számítógépünkön lefordíthatjuk, futtathatjuk és vizsgálhatjuk működését.

Egy feladat megoldására több, különböző elven működő algoritmust hozhatunk létre. Az algoritmusokat összehasonlíthatjuk

- futási idő a bemenet függvényében
- a futás közbeni memóriahasználat
- programkód mérete.

Algoritmusok hatékonyságának, komplexitásának osztályozására a O (*big o, ordó*) jelölést használhatjuk. A definíció alapján egy $f(n)$ függvény $O(g(n))$ halmazbeli, ha léteznek olyan c és N pozitív számok, melyekre $f(n) \leq cg(n)$ bármely $N \leq n$ szám esetén. Az O jelölés segítségével komplexitásuk alapján jellemezni tudjuk az algoritmusokat, egy felső korlátot tudunk mondani a futási időre vonatkozóan. A futási időre vonatkozó felső korlát alapján például egy $O(n^2)$-beli algoritmus $O(n^3)$-beli is.

Egy algoritmus lehet determinisztikus. Az ilyen algoritmusnál bármilyen állapotban is található egyértelmű, hogy mi lesz a következő lépés, amit végrehajt. Az algoritmus véges, ha minden bemenetre a futása véget ér.

Az algoritmusok elemzése a számítástudomány fontos része. Az algoritmusok a programozási nyelv és a hardveres környezettől függetlenül definiálhatóak, elemezhetőek. A bonyolultságelmélet az a terület, ami ilyen szempontok alapján foglalkozik egy algoritmussal.

Szoftvertervezés és -fejlesztés

A hagyományos szoftverfejlesztési folyamat az adat, vagy a folyamat orientált megközelítést használja. Adat orientált megközelítés esetén a szoftver tervezés során az információ ábrázolásán és az adatok belső összefüggéseinek a felderítésén van a fő hangsúly. Az adatot használó, feldolgozó metódusoknak kevesebb szerepe van ez esetben. A folyamat orientált tervezés során – ellentétben a korábbival – a szoftvertervezés elsősorban a szoftver metódusok létrehozásával foglalkozik, az adatnak itt kisebb szerep jut csak.

Egy harmadik programozási módszertan, a manapság már igen széles körben ismert és bizonyítottan hatékony objektum orientált szemlélet (*object-oriented programming, OOP*). Az objektum orientált szoftverfejlesztési folyamatban sokkal hatékonyabban lehet a bonyolult szoftverfejlesztési kérdéseket kezelni, mint a korábbi adat-, vagy folyamat orientált tervezési módszertanokban. Ez a hatékonyság abból ered, hogy objektum orientált megközelítés esetén az adat és a folyamat is hasonlóan fontos szerepet kap, a két részterület nincs fontosság szempontjából megkülönböztetve. Objektumok segítségével egyben kezeljük az összetartozó adatokat és azokat a metódusokat, melyek ezeket az adatokat használják. Az objektumok hierarchikus kapcsolódásainak feltérképezése és megtervezésével a szoftverfejlesztés gyakorlatilag a való világ kapcsolódásainak lemodellezését jelenti. Az objektum orientáltság fő előnyei az absztrakció (*abstraction*) és az egységbe zárás (*encapsulation*) és az osztályok hierarchiába rendezésének a lehetősége.

Mielőtt egy program megszületik, pontosan ismernünk kell azokat a tevékenységeket, amiket el kell végezni és implementálni kell ahhoz, hogy a program elvégezze a kitűzött feladatokat. Az implementálás előtt a program részeket és azok kapcsolatait meg kell tervezni. Minél nagyobb, összetettebb egy program, annál részletesebb tervezésre van szükség. A tervezés során a vezérlésre és a felhasznált programmodulokra vonatkozóan különböző döntéseket hozunk.

Objektum orientált programozás

Az ember a programozás és a valós világ kezelése céljából modelleket épít. Az objektum orientált szoftverfejlesztés közben is a modellek építése és a kapcsolódások feltérképezése a legfontosabb feladat.

Objektum orientált szoftverfejlesztésnél vannak olyan modellezési alapelvek, melyek segítik a modell felépítését. Az absztrakció elve alapján a cél a valós világ leegyszerűsítése olyan szintig, hogy csak a lényegre, a modellezési cél elérése érdekében szükséges részekre összpontosíthassunk. Ebben a részben elhanyagoljuk azokat a részeket, melyek nem fontosak, nem lényegesek a cél szempontjából. Ahogy a programozási technika nevében is szerepel, a fő hangsúly az objektumokon van. Az objektum a modellezett világ egy önálló részét, egységét jelenti. Az objektum tartalmazza azokat a tulajdonságokat és értékeket, ami a modellben leírja az objektum viselkedését. Az objektumnak van egy belső állapota, struktúrája és egy felülete amit a külvilág felé mutat.

Az objektum orientált szoftverfejlesztés során az objektumokat nem egyesével kezeljük (legalábbis a modell építés során), hanem kategorizálva, osztályokként tekintünk rájuk. Egy osztályba a hasonló tulajdonsággal rendelkező objektumokat soroljuk. Az osztály tartalmazza azokat a tulajdonságokat, melyek az objektum viselkedését, működését leírják.

Az OOP esetén az öröklés egy olyan modellezési eszköz, mely segítségével egy alap osztályból új osztályokat hozhatunk létre (származtatott osztály). Az öröklés során az új osztály rendelkezhet az alap osztály bizonyos tulajdonságaival. Vannak olyan osztályok, melyekből nem hozható létre objektum. Az ilyen osztályt absztrakt osztálynak nevezzük, szerepe az öröklési hierarchiában az attribútumai és metódusai örökölhetőségében van.

Szoftvertechnológia

A szoftvertechnológia (*software engineering*) a szoftver fejlesztése, üzemeltetése és karbantartásával foglalkozik. A szoftvertechnológia mérnöki eljárásokat, ajánlásokat adhat a szoftverüzemeltetés és szoftverfejlesztés kapcsán jelentkező kérdések kapcsán.

Régebben a számítógépes programok egyszerűek és kis méretűek voltak. Általában egyetlen programozó, vagy egy kis méretű csapat át tudta látni a feladatot. A hardvertechnológia fejlődésével egyre nagyobb lélegzetű, komplexebb problémák váltak a számítógép által megoldhatóvá. A komplex feladatok a sikeres teljesítés érdekében tervezést, előkészítést, összetett munkaszervezést igényelnek. A mai valós feladatok, szoftverfejlesztési projektek teljesítése során a feladat méretei miatt nagy méretű programozói csoportok dolgoznak a megoldásukon általában egymással párhuzamosan. Egy ilyen projekt során a folyamatosan változó környezeti feltételek miatt szükséges a jól előkészített, szervezett, összehangolt, ellenőrzött és dokumentált munka. Az erre a célra alkalmazott módszerekkel foglalkozik szoftvertechnológia.

Szoftvertechnológiát 1976-ban Boehm a következőképpen definiálta: „Tudományos ismeretek gyakorlati alkalmazása számítógépes programok és a fejlesztésükhöz, használatukhoz és karbantartásukhoz szükséges dokumentációk tervezésében és előállításában.” Az IEEE mérnököket összefogó szervezet is definiálta a szoftvertechnológia fogalmát: „Technológiai és vezetési alapelvek, amelyek lehetővé teszik programok termékszerű gyártását és karbantartását a költség és határidő korlátok betartásával.”

Nehéz egységesen kezelni, összefogni a különböző szoftvertermékek fejlesztéséhez szükséges szoftvertechnológiai módszereket. Sokféle szoftvertechnológiai modell és eljárás született különböző projektek kapcsán. A szoftvertechnológia a következő alaptevékenységeket jelenti általában:

1. A vevői /megrendelői elvárások összegyűjtése, elemzése
2. A megoldás vázlatának, tervének elkészítése – modell, absztrakció
3. Implementálás, kód előállítása
4. Telepítés és tesztelés
5. Karbantartás – folyamatos fejlesztés.

Természetesen vannak még olyan tevékenységek, amik kapcsolódnak, vagy beletartoznak az előbbi felsorolásba. Többek között ilyen tevékenység lehet a projekt menedzsment, minőségbiztosítás, vagy a termék támogatás.

A szoftverfolyamat modellek a szoftver előállításának lépéseire adnak eligazítást (milyen lépésekben kell előállítani a szoftvert az adott környezetben). Nem mondhatjuk hogy a követett modell az egyetlen megoldás. Az előnyöket és hátrányokat mérlegelve a választhatjuk egyiket vagy másikat optimális megoldásként. Ezek a modellek közül néhány fontosabb a következő:

1. Vízesés modell
2. Evolúciós modell
3. Boehm féle spirál modell
4. Gyors prototípus modell
5. Komponens alapú (újrafelhasználás)
6. V modell
7. RUP (Rational Unified Process)

A számítógéppel támogatott szoftvertervezés (*Computer-Aided Software Engineering - CASE*) használt szoftvereket nevezzük CASE-eszközöknek. CASE eszközök a következő lépéseket támogatják:

- Követelményspecifikáció: grafikus rendszermodellek, üzleti és domain modellek
- Elemzés/tervezés**:** adatszótár kezelése, felhasználói interfész generálását egy grafikus interfészleírásból, a terv ellentmondás mentesség vizsgálata
- Implementáció során: automatikus kódgenerálás, verziókezelés
- Szoftvervalidáció során**:** automatikus teszt-eset generálás, teszt-kiértékelés
- Szoftverevolúció során: forráskód visszafejtés (*reverse engineering*); régebbi verziójú, programnyelvek automatikus újrafordítása újabb verzióba.

Mindegyik lépésnek része az automatikus dokumentumgenerálás és a projektmenedzsment támogatás.

UML

Az UML (*Unified Modeling Language*) egy egységes modellező nyelv (http://www.uml.org), amit az Object Management Group hozott létre. Az UML egy olyan eszköztár, amelynek segítségével jól érthető/kezelhető a szoftverrel szemben támasztott követelmények, a szoftver felépítése és a szoftver működése. Az UML grafikus elemeket tartalmaz, mellyel támogatja a szoftver fejlesztés fázisait. Az UML manapság elfogadott és támogatott leíró eszköz, hiszen számos szoftver nyújt lehetőséget az UML használatához.

Az UML diagramokat használ a modell elemek leírására. Ezek két fő csoportba sorolhatók: statikus és dinamikus diagramok. A statikus diagramok a szerkezetét, a dinamikusak a viselkedését modellezik a rendszernek.

- Osztálydiagram: az állandó elemeket, azok szerkezetét és egymás közötti logikai kapcsolatát jeleníti meg.
- Objektumdiagram: az osztálydiagramból származó rendszer adott pillanatban való állapotát jeleníti meg.
- Csomagdiagram: a csomagok más modellelemek csoportosítására szolgálnak, ez a diagram a köztük levő kapcsolatokat ábrázolja.
- Összetevő diagram: a diagram implementációs kérdések eldöntését segíti. A megvalósításnak és a rendszeren belüli elemek együttműködésének megfelelően mutatja be a rendszert.
- Összetett szerkezeti diagram: modellelemek belső szerkezetét mutatja.
- Kialakítás diagram: futásidejű felépítését mutatja. Tartalmazza a hardver és a szoftverelemeket is.
- Tevékenységdiagram: A rendszeren belüli tevékenységek folyamatát jeleníti meg.
- Használati eset diagram: A rendszer viselkedését írja le, úgy, ahogy az egy külső szemlélő szemszögéből látszik.
- Állapotautomata diagram: objektumok állapotát és az állapotok közötti átmeneteket mutatja
- Kommunikációs diagram: objektumok hogyan működnek együtt a feladat megoldása során, hogyan hatnak egymásra.
- Sorrenddiagram: üzenetváltás időbeli sorrendjét mutatja.
- Időzítés diagram: kölcsönhatásban álló elemek állapotváltozásait vagy állapotinformációit írja le.

C++ programozási nyelv

A C programozási nyelvet Dennis Ritchie az 1970-es évek elején fejlesztette ki (Ken Thompson segítségével) a UNIX operációs rendszereken való használat céljából. Ma már jóformán minden operációs rendszerben megtalálható, és a legnépszerűbb általános célú programozási nyelvek egyike, rendszerprogramozáshoz és felhasználói program készítéséhez egyaránt jól használható. Az oktatásban és a számítógépes tudományokban is jelentős szerepe van.

A C++ egy általános célú, magas szintű programozási nyelv. Támogatja a procedurális-, az objektumorientált- és a generikus programozást, valamint az adatabsztrakciót. Napjainkban szinte minden operációs rendszer alá létezik C++ fordító. A nyelv a C hatékonyságának megőrzése mellett törekszik a könnyebben megírható, karbantartható és újrahasznosítható kód létrehozására. Ez azonban sok kompromisszummal jár, amire utal, hogy általánosan elterjedt a mid-level minősítése is, bár szigorú értelemben véve egyértelműen magas szintű (wikipedia.hu).

Ebben a részben a teljesség igénye nélkül a C++ programozási nyelv néhány jellemzőjét soroljuk fel. A jegyzetnek nem célja a C és C++ nyelvek alapos bemutatása. Az adatstruktúrák és algoritmusok jegyzet használatához ismerni kell

- az elemi adattípusokat, struktúrákat, uniót, osztályok alapján létrehozható összetett adatszerkezeteket, a vezérlési szerkezeteket,
- osztályok, konstruktor, destruktor, másoló konstruktor,
- dinamikus memóriakezelés, memória foglalás felszabadítás, mutatók használata, referencia típus,
- tömbök és mutatók kapcsolata,
- függvények, paraméter-átadás működése, függvények átdefiniálása (*overloading*)
- konstansok és makrók
- sablonok.

Dinamikus tömb és láncolt lista

A következő fejezetekben az elemi adatstruktúrákat mutatjuk be, úgy mint a verem, sor, hasító tábla. Ezekben az adatstruktúrákban az adat valamilyen alapelvek alapján kerül tárolásra. Az adat tároláshoz általában dinamikus tömböt, vagy láncolt listát használhatunk. A fejezet első részében ezért a C++ programozási nyelv tömbkezelési lehetőségeinek rövid ismertetése, majd a láncolt listák bemutatása következik.

Mielőtt egy adatstruktúrát létrehozunk, el kell döntenünk, hogy dinamikus tömböt, vagy láncolt listát szeretnénk az adatok tárolásához használni. Emiatt a tömböt és a láncolt listát hívhatjuk alap adatstruktúrának is. A későbbiekben bemutatandó absztrakt adat struktúrák (verem, sor, fa, gráf) a tömböt, és/vagy a láncolt listát használják. Lehetőségünk van egy verem létrehozására tömb és láncolt lista segítségével is. Persze az alap adatstruktúra kiválasztása befolyásolhatja az absztrakt adatstruktúra használhatóságát és hatékonyságát.

Tömb és dinamikus tömb

A tömb a legegyszerűbb és legelterjedtebb eszköz összetartozó adatok együttes kezelésére. A C++ nyelvben a tömbök és a mutatók összekapcsolódnak. Tömbök elérhetőek mutatókon keresztül, mutatók segítségével memóriaterületet tudunk a mutatóhoz rendelni és ezek után tömbként kezelni.

A C++ nyelvben az *N* méretű tömböt 0 és *N*-1 egészekkel indexelhetjük. A tömb mérete statikus, fordítási időben jön létre, hacsak nem dinamikusan a programozó gondoskodik a tömbterület lefoglalásáról majd felszabadításról.

A dinamikus memóriahasználat során, hogy adataink számára csak akkor foglalunk memóriaterületet, amikor szükség van rá, ha pedig feleslegessé válik a lefoglalt memóriaterület, akkor azonnal felszabadítjuk azt. Ez azért fontos a C és C++ programozási nyelvekben, mivel itt nincs olyan „szemétgyűjtő” (*garbage collector*) mint a JAVA -ban vagy C# -ban. A felszabadítatlan memória „memóriaszivárgáshoz” (*memory leak*) vezet.

A C++ -ban lehetőség van használni a C malloc és free utasítását, memória terület foglalására és felszabadítására, azonban használhatjuk az erre létrehozott C++-os operátorokat is. A new operátor az operandusban megadott típusnak megfelelő méretű területet foglal a memóriában, és egy arra mutató pointert ad vissza. A delete felszabadítja a new által lefoglalt memóriaterületet:

Egy mutató lefoglalása és felszabadítása a következő kódsorokkal végezhető el C++ programozási nyelven:

```
int *p = new int;
delete p;
```

Több egymás után elhelyezkedő elem számára is foglalhatunk területet. Ezt az adatstruktúrát nevezzük dinamikus tömbnek. A dinamikus tömb lefoglalásához és felszabadításához a következő kódsorokat kell végrehajtani:

```
int *v = new int [123];
delete[] v;
```

Többdimenziós tömböt dinamikus lefoglalása esetén a dimenziók számának megfelelően lépésenként le kell foglalni a mutatókat. A következő példában két dimenziós mátrixot foglalunk és szabadítunk fel mutatók segítségével:

```
int **T = new int * [17];
for(int i = 0;i < 17;i++) {
    T[i] = new int [10];
    for(int j = 0;j < 10;j++) {
        T[i][j] = ERTEK;
    }
}
for(int i = 0;i < 17;i++){
    delete[] T[i];
}
delete[] T;
```

Láncolt lista

A tömbök segítségével sok esetben meg lehet oldani az összetartozó adatok tárolását, azonban megvannak ennek az adatstruktúrának is a hátrányai:

- a tömb méretét előre ismernünk kell (kivétel dinamikus tömbök)
- a már lefoglalt, létrehozott tömb mérete nem módosítható
- a memóriában folytonosan kerül tárolásra, ha egy új elemet szeretnénk a tömbbe beszúrni, a tömb egyik felében szereplő elemek mozgatásához vezet.

Ezeket a hátrányokat a láncolt lista adatszerkezettel elkerülhetjük. Láncolt lista segítségével lefoglalt tömb mérete miatti korlátokat elkerülhetjük, a lista hosszára csak a rendelkezésre álló szabad memória mennyisége ad korlátot. A láncolt lista egy eleme két részből épül fel. Egyrészt tartalmazza a tárolni kívánt adatot, vagy adatokat és tartalmaz egy olyan mutatót, ami a lista egy másik elemét mutatja. A láncolt lista a dinamikus tömbhöz képesti hátránya a közbülső elemek elérhetőségéből ered. Míg egy tömb esetén ha tudjuk, hogy a *k*. elemet szeretnénk elérni, akkor a tömb indexelésével rögtön hozzáférhetünk ehhez az adathoz, addig a láncolt listában a lista elejéről indulva a mutatókon keresztül addig kell lépkedni, míg a *k*. elemhez nem értünk. A véletlenszerű lista elem megtalálása a lista hosszával arányos időt igényel.

Láncolt listákat gyakran alkalmazzuk összetettebb adatstruktúrák (verem vagy sor) építésekor. A láncolt lista adatmezői tetszőleges adatot tárolhatnak, akár mutatókat más adatszerkezetekre, vagy több részadatból felépülő struktúrákat. Több mutató használatával nemcsak lineáris adatszerkezeteket tudunk építeni, hanem tetszőleges elágazó struktúrát is.

Egyszeresen láncolt lista

Egyszeresen láncolt listában egy darab mutató jelöli a lista rákövetkező elemét. Ha ismerjük a lista legelső elemét (lista feje), akkor abból elindulva a mutatók segítségével végigjárhatjuk a listában tárolt elemeket. A lista legutolsó elemének mutatójának értéke NULL, ez jelzi, hogy tovább nem tudunk haladni a listában. Láncolt lista esetén általában egyszeresen láncolt listára gondolunk.

Egy egész számokat tároló láncolt listát például a következő osztállyal valósíthatjuk meg:

```
class IntNode{
```

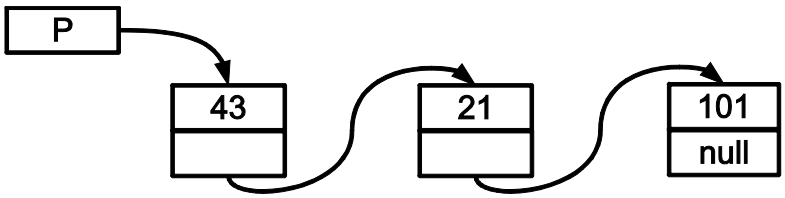
```
public:
    IntNode() {
        next = 0;
    }
    IntNode(int i, IntNode *new_node = 0) {
        data = i; next = new_node;
    }
    int data;
    IntNode *next;
};
```

Minden egyes lista elem (IntNode) a data tagban tárolja az adatot, a next mutatja a lista következő elemét. Az IntNode osztály két konstruktort is tartalmaz. Az első NULL értékre állítja a next mutatót és az adat tag értékét nem definiálja; a második két paramétert használ, az első paraméterrel a data adattagot, a másodikkal a next mutató értékét inicializálja.

Az IntNode osztály segítségével hozzunk létre egy három elemet tartalmazó láncolt listát:

```
IntNode *p = new IntNode(43);
p->next = new IntNode(21);
p->next->next = new IntNode(101);
```

A 43, 21 és 101-et tartalmazó láncolt lista az 1. ábrán látható. A p mutató jelöli a lista első elemét (fejét, *head*).



1. ábra: Láncolt lista három elemmel

Ily módon a láncolt listát akármeddig bővíthetjük (ameddig tudunk új elemet foglalni a memóriában). Azonban ez a módszer a lista bővítésére egy bizonyos hossz után nehézségekhez vezet, hiszen például egy 100 elemű lista utolsó elemének a létrehozásához 99 hosszúságban kellene a next-eket egymás után felfűznünk, majd végigjárni. Ezt a kellemetlenséget elkerülhetjük, ha nemcsak a lista első elemét, hanem az utolsót is számon tartjuk (*tail*). Ehhez az újfajta listához hozzunk létre egy IntList nevű osztályt, ami magát a listát kezeli.

```
class IntList {
public:
    IntList() {head = tail = 0;}
    ~IntList();
    bool isEmpty() {return head == 0;}
```

```
    void addToHead(int);
    void addToTail(int);
private:
    IntNode *head, *tail;
};
```

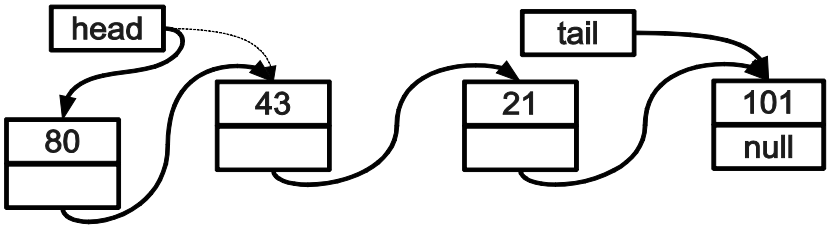
Az IntList osztály két adattagot tartalmaz, két mutatót, melyek a lista első és a lista utolsó elemeire mutatnak (head, tail). A lista osztály tartalmaz két olyan metódust, mellyel a lista elejére (addToHead) vagy a végére (addToTail) tudunk új elemet beszúrni. A láncolt lista a következő utasítással hozható létre:

```
IntList L;
```

Beszúrás láncolt listába

Ha van egy láncolt listánk valószínűleg a lista bővítése az egyik legfontosabb feladat, amit végre kell hajtanunk. A lista bővítés lépései kis mértékben különböznek egymástól attól függően, hogy a lista melyik pozíciójába szeretnénk az új lista elemet beszúrni. Ha a láncolt lista aktuális legelső eleme elé szeretnénk beszúrni egy új elemet, akkor a következő lépéseket kell végrehajtani:

1. Egy új csúcs létrehozása/lefoglalása
2. Értékadás az új lista csúcs adat tagjának
3. Mivel a lista első eleme lesz az új csúcs, ezért az új csúcs next tagjának értéke a lista korábbi első csúcsának címe lesz, ami a head mutató aktuális értékével egyenlő
4. A head-nek a lista új csúcsára kell mutatnia, hogy a későbbiekben is elérjük a lista első elemét.



2. ábra: Az előző láncolt lista első helyére új elem beszúrása

A 2. ábrán az előző lista első pozíciójába láncoltuk be a 80-as listaelemet. Az ábrán a szaggatott vonallal ábrázolt nyíl jelzi azt a mutatót, amit a beszúrás során megváltoztatunk. Az első helyre beszúrást a következő osztály-metódussal végezhetjük el:

```
void IntList::addToHead(int new_data) {
    head = new IntNode(new_data, head);
    if(tail == 0)
        tail = head;
}
```

A beszúrás műveletre az IntNode osztály konstruktorát hívtuk segítségül, mely az új memóriaterület lefoglalása után a korábbi head mutató értékére irányítja az újonnan lefoglalt

IntNode objektum next mutatóját. Abban az esetben, ha láncolt lista még nem tartalmazott egyetlen egy elemet se korábban, gondoskodni kell hogy a tail mutató is helyes lista elemre mutasson a művelet végrehajtása után (tail = head).

Láncolt lista utolsó helyére való beszúrás első két lépése nem változik az első helyre való beszúráshoz képest, ugyanúgy le kell foglalni a megfelelő memóriaterületet az új elem számára, majd inicializálni kell az adatrészt. Az utolsó helyre való beszúrás lépései a következőek:

1. Egy új csúcs létrehozása/lefoglalása
2. Értékadás az új lista csúcs adat tagjának
3. Mivel az új csúcs lesz a lista utolsó eleme, ezért az új csúcs next-jének értéke NULL
4. Az új csúcsot a korábbi utolsó csúcs next-jének az új csúcsra való állításával befűzzük a listába (ez a csúcs a tail aktuális értéke)
5. A tail-nek a lista új csúcsára kell mutatnia.

Az utolsó helyre beszúrást a következő osztály-metódussal végezhetjük el:

```
void IntList::addToTail(int new_data) {
    if(tail != 0) {
        tail->next = new IntNode(new_data);
        tail = tail->next;
    } else
    head = tail = new IntNode(new_data);
}
```

Ha a lista már tartalmaz legalább egy elemet, akkor a tail mutató után kell befűzni az új lista elemet, majd frissíteni kell a tail mutató értékét (tail = tail->next). Ha még üres volt a lista, akkor a lista végére való beszúrás gyakorlatilag megegyezik a lista elejére való beszúrással, használhatnánk akár az addToHead metódust is a művelet végrehajtására.

Láthatjuk, hogy a beszúrás művelet head és tail mutatókat tartalmazó egyszeresen láncolt lista esetén konstans időben ($O(1)$) elvégezhető. Ugyanakkor, ha nem használnánk tail mutatót az utolsó elem jelölésére, akkor a lista elejére való beszúrás konstans időt igényel, de ha a lista utolsó helyére akarjuk beszúrni az új lista tagot, akkor végig kell menni az egész listán, hogy megtaláljuk az utolsó lista elemet. Ebben az esetben a lista hosszával arányos a beszúrás időigénye ($O(n)$).

Törlés láncolt listából

Hasonlóan a beszúrás művelethez, a törlés esetén is különböző lépéseket kell a törlendő lista elem helyének függvényében. Akkor vagyunk egyszerűbb helyzetben, ha az első lista elemet és a listában tárolt adattagot akarjuk törölni az egyszeresen láncolt listából. A törlés végrehajtásához a következő lépéseket hajtsuk végre:

1. Állítsunk egy ideiglenes mutatót a lista első elemére
2. Állítsuk a head mutatót a jelenlegi láncolt lista második elemére (pl. az ideiglenes mutató next-je segítségével megkaphatjuk a második listaelemet)
3. Az ideiglenes mutató memóriaterületét felszabadíthatjuk.

Általában a fenti lépések végrehajtásával elvégezhetjük az első elem törlését, azonban vizsgáljunk meg két speciális esetet. Ha a lista üres, akkor törölni sem tudunk belőle, ezért ezt az esetet külön kell kezelni. A másik speciális eset, amikor egy egy elemű listából törlünk, mert ekkor a törlés után egy üres listát kapunk, tehát a head és a tail mutatót is NULL-ra kell állítani.

A lista utolsó elemének a törléséhez meg kell keresni az utolsó előtti elemet (egyszeresen láncolt lista esetén), mert ennek az elemnek a next mutatóját kell aktualizálnunk a törlés után. Az utolsó előtti elem egy ciklussal kereshető meg, a megállási feltétel ügyes beállításával (tmp->next!=tail). A ciklusban a tmp mutatót léptetjük a head-ből indulva egészen addig, míg a next mutatója az utolsó elemre nem mutat, azaz a tmp az utolsó előtti listaelemre mutat. Ha van utolsó előtti elem, akkor annak megtalálása tehát:

```
for(tmp=head;tmp->next!=tail;tmp=tmp->next)
```

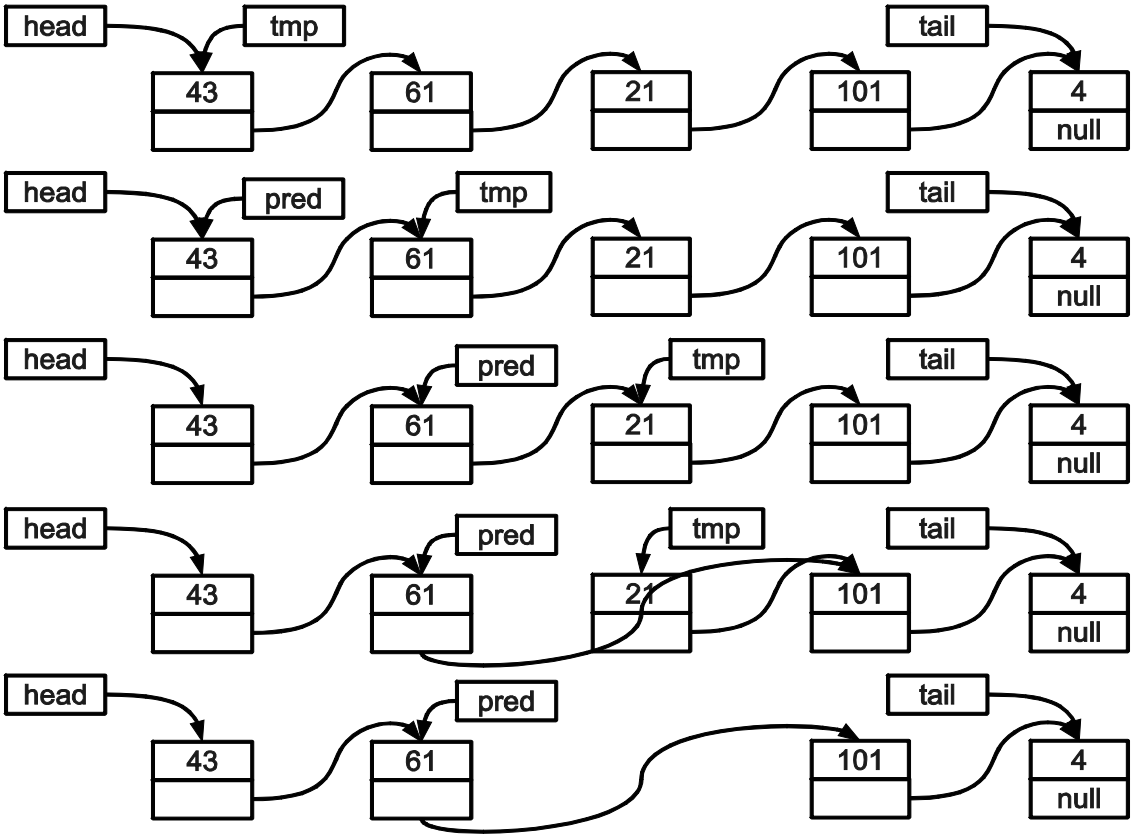
Ha már megtaláltuk az utolsó előtti lista elemet, akkor az utolsó listaelem törlése hasonló lépéseket tartalmaz, mint amiket az első elem törlésekor elvégeztünk. Az elvégzendő műveletek a következőek:

1. Keressük meg a lista utolsó előtte elemét (tmp mutató segítségével)
2. A tmp mutató next-je legyen NULL mutató
3. A tail mutatóhoz tartozó memóriaterületet szabadítsuk fel
4. Állítsuk a tail mutatót a tmp által jelölt memóriaterületre.

Hasonlóan a lista elejéről való törlés esetéhez itt is két speciális esetre kell odafigyelni. Ha a lista üres, akkor az utolsó elem törlését sem tudjuk elvégezni. Ha a lista egy elemet tartalmaz, akkor az utolsó elem eltávolításával egy üres listát kapunk.

Általános esetben nem a lista elejéről, vagy a végéről törlünk, hanem a lista bármely pozíciójában található elemet törölhetjük. Általában egy adott értéket kell megtalálnunk a listában, majd eltávolítani azt belőle. Az eltávolítás azt jelenti, hogy a törlendő elem előtti listaelem next mutatóját átállítjuk az eltávolítandó listaelem utáni elemre, majd elvégezzük a törlést. Ahhoz, hogy a keresett elemet ki tudjuk láncolni a listából, érdemes a keresés során az aktuális elem előtti listaelemet is megjegyezni.

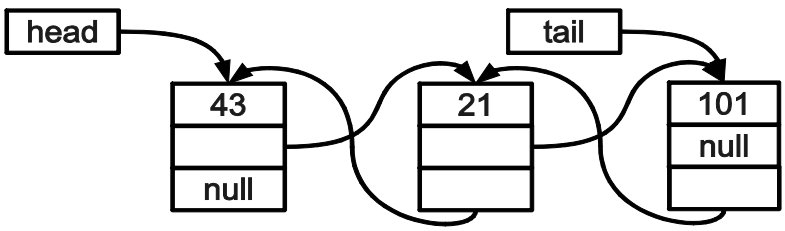
A 3. ábrán egy példát láthatunk láncolt listában való keresés majd a keresett elem törlésére. Tegyük fel, hogy a 21-et tartalmazó elemet szeretnénk megtalálni, majd törölni a listából. Ehhez a tmp és a pred segédmutatókat használjuk. A tmp az éppen aktuálisan megtalált listaelemre mutat, a pred pedig az aktuális előtti elemre. A tmp és a pred mutatókat addig léptetjük, míg megtaláljuk a keresett értéket tartalmazó elemet. Az érték megtalálása után a pred-hez tartozó listaelem következő elemének beállítjuk a tmp következő listaelemét. Ezek után törölhetjük a tmp által mutatott listaelemet. Az egyszeresen láncolt lista C++ implementációja kiegészítve a láncolt listából való törlés műveletekkel a mellékletben található.



3. ábra: Keresés és törlés egyszeresen láncolt listában

Kétszeresen láncolt lista

Az egyszeresen láncolt lista egy listaeleme csak egy mutatót tartalmaz, ami a lista következő elemének címét jelöli, így közvetlenül nem lehet megmondani hogy mi volt az előző elem. Már az eddigiek alapján is láthattuk, hogy van olyan lista művelet (pl. lista végéről való törlés), amikor az előző listaelem mutatóját kell módosítani. Ha a listaelem az előző és a következő elemre is tartalmazna egy-egy mutatót, akkor ezzel bizonyos láncolt lista műveletek egyszerűsödnek, ugyanakkor a két mutató miatt természetesen vannak olyan műveletek is, melyek emiatt bonyolultabbak lesznek. Az ilyen listákat kétszeresen láncolt listának (*doubly linked list*) nevezzük. A 4. ábrán egy kétszeresen láncolt listát láthatunk, melyben a 43, 21 és 101 értékeket tároltuk.



4. ábra: Három elemet tartalmazó kétszeresen láncolt lista

A kétszeresen láncolt lista implementálása hasonlóan történhet, mint az egyszeresen láncolt változat. A lista adattagoknál egy előre mutató (next) és egy hátra mutató (prev) pointert is létre

kell hoznunk és új elem beillesztésekor gondoskodni kell, hogy mind az előre, mind a hátrafelé mutatók megfelelően változzanak. A következő kódrészletben egy kétszeresen láncolt lista implementációját láthatjuk.

```
class IntDNode{
public:
    IntDNode() {
        next = prev = 0;
    }
    ...
    int data;
    IntDNode *next;
    IntDNode *prev;
}
```

Körkörösen láncolt lista

Bizonyos esetekben körkörösen láncolt listát (*circular list*) kell megvalósítanunk. Ebben a listában a listaelemek egy kört alkotnak, az első és az utolsó elem össze van kötve egymással. A kör miatt minden elemnek van egy rá következője. Ilyen példát találhatunk a processzor ütemezésekor, amikor minden folyamat ugyanazt az erőforrást használja, az ütemező azonban nem futtat egy folyamatot egészen addig míg minden előtte levő nem került kapott processzoridőt. Egy ilyen listában, aminek nincs kitüntetett első, vagy utolsó eleme, elég bármely elemét ismernünk, hogy ezután az elemen keresztül hozzáférjünk az összes többihez, mindössze arra kell figyelni a láncolt lista végigjárásakor, hogy melyik volt az az elem, ahonnét a bejárást indítottuk.

Őrszemes lista

A lista műveletek (pl. törlés) egyszerűsödne, amennyiben a lista végein levő elemeknél nem kellene ellenőrzéseket tennünk. Ezt megvalósíthatjuk úgy hogy speciális listaelemeket (őrszemeket) vezetünk be a lista határainak jelölésére. Ezek az őrszemek ugyanolyan felépítésűek, mint a lista bármely eleme, fizikailag a láncolt listában szerepelnek, azonban logikailag nem részei a listának. Őrszemek segítségével a null mutatók kezeléséből eredő ellenőrzéseket elkerülhetjük.

Ritka mátrix tárolása láncolt listával

Sokszor táblázatos formában, többdimenziós tömbben tárolunk mátrixokat. Azonban ha a mátrix kevés értéket tartalmaz memóriahasználat szempontjából gazdaságtalan a tömb használata. Az ilyen mátrixokat ritka mátrixnak hívjuk (*sparse matrix*).

Nagyméretű mátrix tárolására a sorok és oszlopok számának szorzata határozza meg a tároláshoz szükséges tömb méretét. Akkor, ha ennél a szorzatnál lényegesen kisebb azoknak a celláknak a száma, ahol adatot tárolunk, akkor érdemesebb egy láncolt listát használni tömb helyett. A láncolt listában tárolni kell a cella értékét és az értékhez tartozó indexeket. A láncolt lista segítségével lényegesen csökkenthetjük a mátrix tárolására szükséges memóriahasználatot.

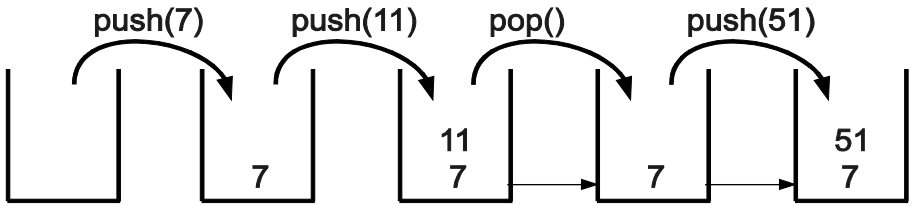
Verem adatszerkezet

A verem (*stack*) egy olyan lineáris adatstruktúra, melybe csak az egyik végén lehet adatot berakni, vagy adatot kivenni belőle. Verem jellegű tárolási struktúrával nem csak a számítástechnikában találkozhatunk, hanem mindennapokban is sok esetben verem jelleggel végzünk műveleteket (pl. étkezőszekrényben tányérokat egymásra rakva tároljuk, a legfelsőt tudjuk csak kivenni és a legtetejére tudunk új tányért berakni). A verem amiatt, hogy az utoljára berakott elemet lehet belőle először kivenni egy LIFO (*last in first out*) jellegű adatszerkezet. Verem adatszerkezetet a számítástechnikában sok helyen használjuk: operációs rendszerek, függvények hívásánál egy verem területre kerül a hívó függvény paraméter listája, lokális változói és visszatérési címe.

Ahhoz, hogy a verem adatszerkezetet használni tudjuk definiálni kell néhány műveletet, amit értelmezni tudunk verem esetén. A verem használatához és állapotainak lekérdezéséhez a következő műveletekre van szükségünk:

- Clear() – verem ürítése
- isEmpty() – leellenőrzi, hogy üres-e a verem
- Push(i) – az i elemet a verem tetejére teszi
- Pop() – kiveszi a legfelső elemet a veremből és visszatér annak értékével

A következő példában push és pop műveletek egy sorozatát hajtjuk végre, a verem aktuális tartalmát a 5. ábrán láthatjuk. A verembe először berakjuk a 7, majd a 11 számokat. A verem tetején levő szám eltávolítása után berakjuk az 51-et. Amivel végeredményben a verem a 7 és 51 elemeket tartalmazza.



5. ábra: A verem tartalmának változása push és pop műveletek hatására

A verem nagyon hasznos adatszerkezet, általában akkor van rá szükségünk, ha a tároláshoz képest fordított sorrendben van szükség az adatokra. Verem alkalmazására egy mintafeladat lehet egy matematikai kifejezésben a zárójelek nyitó és záró felének a párosítása. A zárójel párosítás feladat könnyen megoldható egy verem segítségével. A kifejezés balról jobbra való olvasásával nyitó zárójel esetén a zárójelet a verem tetejére tesszük (Push), bezáró zárójel esetén pedig kiolvasunk (pop) egy elemet a verem tetejéről. A kiolvasott zárójelnek ugyanolyan fajtájúnak kell lennie, mint a bezáró volt. A kifejezés hibás, ha a kivett zárójel nem ugyanolyan fajtájú, vagy üres veremből próbálunk meg kiolvasni, vagy ha az ellenőrzés végén a veremben maradt valami.

Ha egy üres veremből akarunk valamit kivenni a verem tetejéről (pop), akkor általában az üres verem hiba szokott jelentkezni. A veremben tárolható elemek számára általában egy felső korlátot szokás adni, ha több elemet teszünk a verembe, mint ez a felső korlát, akkor a tele verem hibaüzenetet kapjuk (*stack overflow*).

Verem implementálása tömbbel

A következő példaprogramban egy egészek tárolására alkalmas vermet hozunk létre. A verem dinamikus tömböt használ az elemek tárolására. A verem maximális méretét/elemszámát a létrehozáskor határozzuk meg.

```
class IntStack {
public:
    IntStack(int cap) {
        top = 0; capacity = cap;
        data = new int[capacity];
    }
    ~IntStack() {
        delete [] data;
    }
    bool isEmpty() const {return top==0;}
    void Push(int i);
    int Pop();
    void Clear();
private:
    int *data;
    int capacity;
    int top;
}
```

Az IntStack osztály a data dinamikus tömbben tárolja a benne szereplő elemeket. A dinamikus tömböt a konstruktor hozza létre, a destruktor szabadítja fel. A verem a dinamikus tömbbel kapcsolatban tudja, hogy összesen mennyi elem fér bele (capacity) és jelenleg mennyi elem van benne, azaz hogy hányadik indexig tartalmaz a tömb tárolt értékeket (top). Az isEmpty() metódussal azt lehet lekérdezni, hogy a verem tartalmaz-e már valamit.

A Pop() metódus a verem tetején levő elemmel tér vissza, miközben a verem tetejéről lekerül ez a visszaadott érték. Fizikailag benne maradhat a tömbben a visszaadott érték, azaz nem kell a tömb értékét módosítani, hiszen elég ha a top változó módosításával tudjuk, hogy a veremnek már nem része a legutóbb visszaadott elem. A Pop() metódus a következőképpen épülhet fel:

```
int IntStack::Pop() {
    if(top > 0) {
        top--;
        return data[top];
    } else {
         cout <<"Stack is empty" << endl; //ures verem hibauzenet
         return -1;
    }
}
```

A Push() metódus a paraméterként kapott új értéket helyezi el a verem tetejére. Abban az esetben, ha a verembe még elfér (top<capacity), akkor el tudjuk helyezni a verem tetején. Ha a verem tele van, akkor ezt kivétel dobásával jelezzük (megj.: a kivétel dobás helyett meg lehetne valósítani a data tömb méretének növelését is, melyhez egy új tömb foglalásával, a régi tömb értékeinek másolásával és a régi tömb felszabadításának lépéseivel érhetjük el). A Push() metódus a következőképpen épülhet fel:

```
void IntStack::Push(int i) {
    if(top < capacity) {
        data[top] = i;
        top++;
    } else cout << "Stack is full" << endl; //tele a verem hibauzenet
}
```

A verem tartalmának ürítésére a Clear() metódust használjuk. Az adatszerkezetben a top adattag jelzi, hogy hol van a verem teteje, azaz hány darab elem található benne. A verem tartalmát törölhetjük, ha a top értékét 0-ra állítjuk.

```
void IntStack::Clear() {
    top = 0;
}
```

Verem implementálása láncolt listával

A dinamikus tömb helyett a verem adatstruktúrát láncolt listával is meg lehet valósítani. A láncolt lista előnye, hogy elemeinek száma pontosan annyi, mint amennyi elem a veremben van, nincs szükség nagyobb tárterület előre lefoglalására. A dinamikus tömböt használó megvalósításban ezzel ellentétben előre lefoglalunk egy capacity méretű tárolót; így könnyen előfordulhat, hogy a tároló nagy része üresen áll. Mivel a verem tetején levő elemet tudjuk csak elérni az adatszerkezet jellegéből adódóan, ezért nincs szükség közvetlen címzésre, az adatok láncolt listában való tárolása kézenfekvő választás verem adatszerkezethez.

A következő példaprogramban egy vermet hozunk létre egyszeresen láncolt listával. A verembe egészeket szeretnénk eltárolni:

```
class IntStack {
public:
    IntStack() {...}
    ~IntStack() {...}
    bool isEmpty() const {return data.isEmpty();}
    void Push(int i);
    int Pop();
private:
    IntList data;
}
```

```
void IntStack::Push(int i) {
      data.addToTail(i);
}

int Pop() {
    if(data.isEmpty())
            //hibauzenet
    else return data.deleteFromTail();
}
```

Az IntStack osztályban a korábban bevezetett IntList egészeket tartalmazó láncolt listát használjuk. A láncolt listába a végére pakoljuk be az új értékeket (addToTail()) és csak a végéről vehetünk ki belőle (deleteFromTail()). Ezzel biztosítjuk a LIFO elvet.

Láncolt lista használata esetén az a hiba nem fordulhat elő, hogy a verem megtelik, hiszen a lista csak akkor bővül pontosan egy elemmel, ha új elemet teszünk bele. A veremből kivételkor a listából is rögtön törlődik és felszabadul a kivett elem. Üres verem esetén a láncolt lista sem tartalmaz elemet, ezért az üres veremből/listából való elem kivételt kezelni kell.

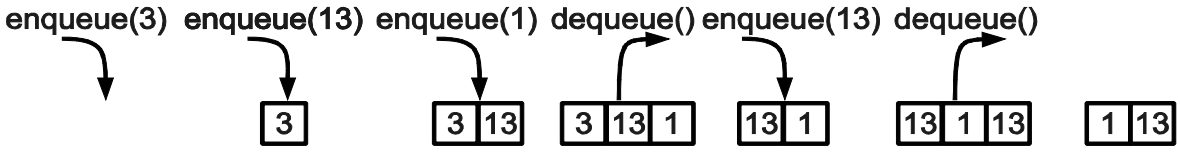
Sor adatszerkezet

A sor (*queue*) nem más, mint egy „várakozási lista”, ami úgy növekszik, hogy elemeket tudunk a végére hozzáadni és úgy csökkenhet, hogy elemeket vehetünk el az elejéről. A veremmel ellentétben a sor olyan adatszerkezet, melynek mindkét végén végezhetünk műveleteket. A sor jellegéből adódóan az utoljára hozzáadott elemnek egészen addig várakoznia kell, míg a korábban hozzáadott elemeket ki nem vettük belőle. Az ilyen adatszerkezetek FIFO (*first in first out*) tulajdonságúak.

A sorhoz a következő alapműveleteket definiálhatjuk:

- clear() - sor ürítése
- isEmpty() - annak ellenőrzése, hogy a sor üres-e
- enqueue(element) – az új elem a sor végére való beszúrása
- dequeue() - visszaadja a sor elején levő elemet.

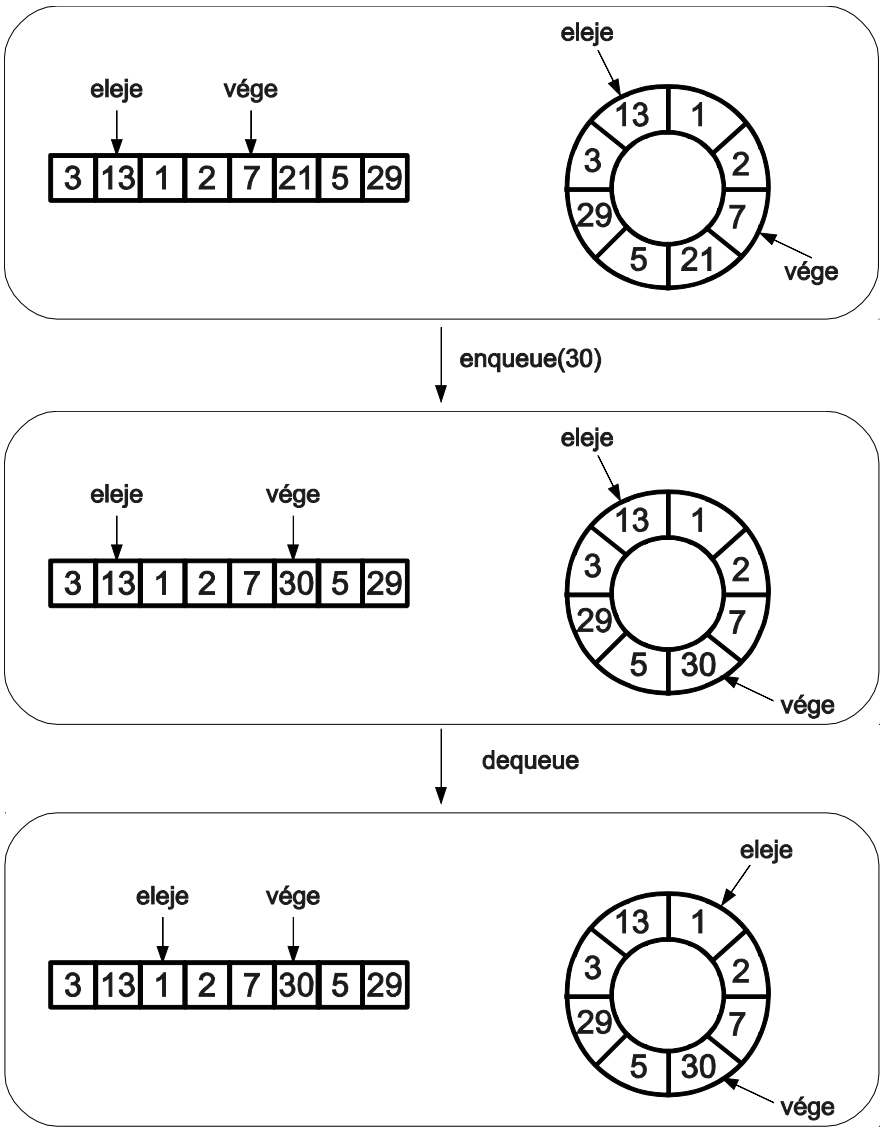
A 6. ábra az enqueue és dequeue műveletek hatását szemlélteti egy egészeket tartalmazó soron. Üres sorból indulva berakjuk a sorba a 3, 13, majd az 1 értékeket. Ezután a sorban a 13 és az 1 érték maradt.



6. ábra: Az enqueue és dequeue műveletek hatása egy sor adatszerkezeten

Összehasonlítva a sort a veremmel láthatjuk, hogy ebben az adatszerkezetben a veremmel ellentétben az adatszerkezet mindkét oldalán történnek adat műveletek. Ugyanúgy, mint a veremnél itt is lehetőségünk van a sorban tárolt adatok tárolására dinamikus tömböt, vagy láncolt listát használni.

Ha a sor adatszerkezetben tárolandó adatokhoz dinamikus tömböt szeretnénk használni, akkor a tömbhöz nyilván kell tartani a sor elejét és végét, azaz az első elem és az utolsó elem indexét is. Ha a kisebb index jelöli a sor elejét, a nagyobb (vagy nagyobb egyenlő) pedig a sor végét, akkor mind a sorból kivétel, mind a sorba berakás esetén valamely tömbindex növekedése történik. Ha valamely index a tömb utolsó elemére mutat, akkor az index növekedése esetén a rá következő érték a tömb legelső eleme lesz. Üres sor esetén a sor eleje és a vége indexek egy soron kívüli értékre mutatnak. Teli sor esetén a sor vége index a sor elejétől balra mutató értéket jelöli, ekkor a sor vége index nem növelhető (a sor vége index balról nem „előzheti le” a sor eleje indexet). A sor adatszerkezet egy úgynevezett „körkörös” tömb segítségével implementálható. A 7. ábrán egy sor adatstruktúrát láthatunk egy nyolc méretű dinamikus tömböt használva. A sor kezdetben a 13, 1, 2 és 7 egészeket tartalmazza. Első lépésben berakjuk a sor végére a 30 értéket, majd a következő lépésben kivesszük a sor elején található elemet.



7. ábra: Sor megvalósítása körkörös dinamikus tömb segítségével

A következő példában C++ nyelven implementálunk egy olyan sort, amiben egészeket szeretnénk tárolni. A tárolásra MAX méretű dinamikus tömböt használunk. A sor a data tömböt használja az adatok tárolására, a beginI és endI változókat használjuk arra, hogy a sor elejét és végét jelöljük.

```
class IntQueue {
private:
    int data[MAX];
    int beginI, endI;
public:
    IntQueue() {
        beginI = endI = -1;
    }
```

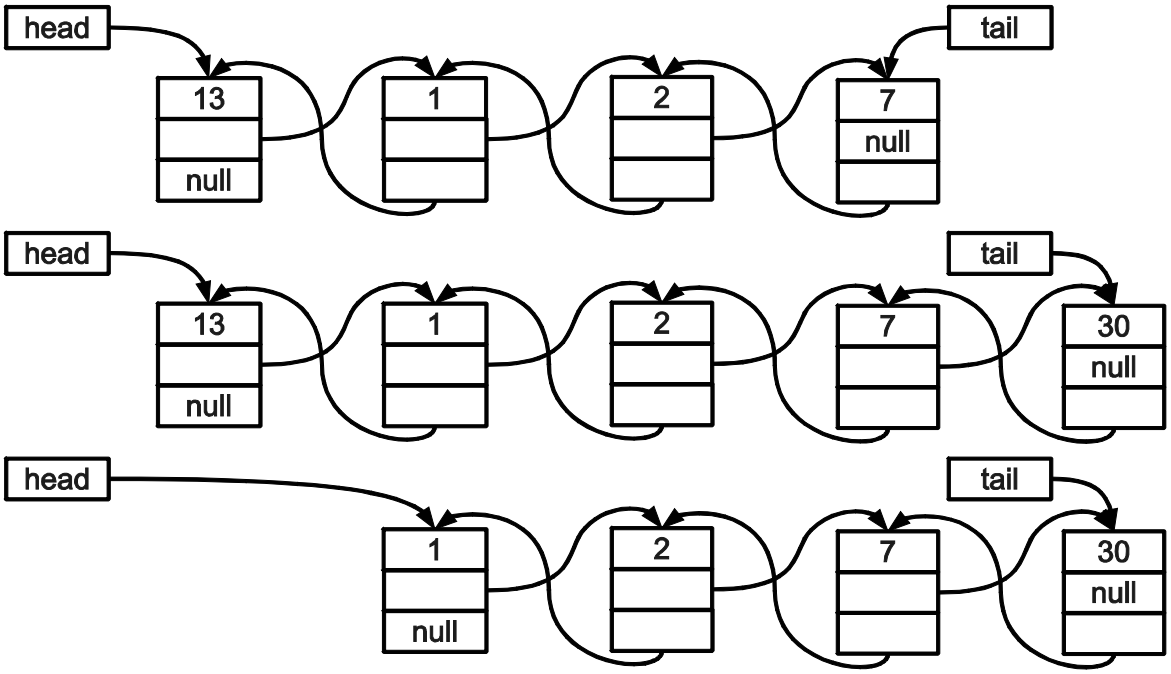
```
    bool isEmpty() const;
    int dequeue();
    void enqueue(int);
};

int IntQueue::dequeue() {
    int tmp = -1;
    if(endI==-1)
        cout << "Queue is empty" << endl;
    else{
        tmp = data[beginI];
        if(beginI==endI)
            beginI = endI = -1;
        else
            beginI = ++beginI % MAX;
    }
    return tmp;
}

void IntQueue::enqueue(int tmp) {
   if(endI == -1){
        beginI = endI = 0;
        data[0] = tmp;
   } else {
        if((endI+1) % MAX == beginI)
            cout << "Queue is full"<< endl;
        else {
            endI = ++endI % MAX;
            data[endI] = tmp;
        }
   }
}
```

Gondoljuk végig, hogy a dinamikus tömbök mellett milyen más adatszerkezetet használhatunk egy sor implementálásához. Ugyanúgy, mint a verem adatszerkezetnél, a sor esetén is a dinamikus tömbök helyett bizonyos esetekben sokkal célszerűbb a sort láncolt lista segítségével (és elsősorban a kétszeresen láncolt listával) implementálni. Az egyszeres láncolt lista esetén is meg lehet valósítani a sort, azonban mivel elején és a végén is műveleteket kell

végrehajtani, ezért a az előző elemekre is szükség van a láncolások elvégzéséhez. Egyszeresen láncolt lista esetén nehézkessé válhat a megvalósítás. Kétszeresen láncolt listára az egyszeresen láncolt helyett, mert a sor tulajdonságai miatt mindkét végén műveleteket kell tudnunk elvégezni. A sor hatékonyságát növelheti, ha bármelyik irányból meg tudjuk mondani az előző elem címét és elhelyezkedését. Láncolt lista használata esetén a sor bővíthetőségének nincsenek korlátai, a dinamikus tömbbel ellentétben itt nem kell előre memóriaterületet foglalni az elemek tárolásához. A 8. ábrán az előzőleg bemutatott dinamikus tömbbel implementált sor és a rajta elvégzett műveletek láthatóak kétszeresen láncolt lista segítségével.



8. ábra: Sor megvalósítása kétszeresen láncolt lista segítségével

Számos számítástechnikai alkalmazást találhatunk sorok használatára. Tipikus alkalmazása a sor adatszerkezeteknek például a processzor várakozási sora, mely azokat a folyamatokat tartalmazza, melyek processzorra, futásra várakoznak. Másik tipikus alkalmazása a sor adatszerkezeteknek az operációs rendszerek esetén két folyamat között az adatok kicserélésére, kommunikációra az operációs rendszer által biztosított tárterület. Ezt a tárterületet puffernek nevezzük és általában sor adatszerkezet segítségével valósítjuk meg. A jegyzetben is találkozhatunk majd olyan algoritmusokkal, például a gráfalgoritmusok között, melyek sor adatszerkezetet használnak működésük során (például a gráfbejárásra használt algoritmus).

Prioritásos (elsőbbségi) sor adatszerkezet

Sok esetben az előző részben ismertetett sor nem elég a feladat megoldásához, mert a FIFO adatsorrendet más elveket figyelembe véve felül kell írni. Ilyen elv lehet például a sorban tárolt adat fontossága, prioritása. A prioritásos sor/elsőbbségi sor (*priority queue*) a FIFO elvet egy prioritással kiegészítve először a magasabb prioritással rendelkező elemeket szolgáltatja. Az azonos prioritással rendelkező elemek között a FIFO elv érvényesül. Ilyen példát találhatunk például egy kórház sürgősségi osztályán, ahol a súlyosabb sérülteket (magasabb prioritás) látják el először, és csak utána gondoskodnak az enyhébb sérültekről.

A prioritásos sorhoz nehéz olyan hatékony implementációt találni, melybe viszonylag gyorsan lehet elemet berakni és kivenni. Mivel berakásra az elemek eltérő prioritással érkeznek, ezért kivétel esetén nem feltétlenül a sor legelső eleme lesz az, aminek a legnagyobb a prioritása és ki kell venni a listából. Alapvetően kétfajta módon lehet prioritásos sort létrehozni láncolt lista, vagy dinamikus tömb segítségével.

- Amikor berakunk egy új elemet, akkor a prioritást figyelembe véve szúrjuk be, azaz a prioritásnak megfelelő sorrendben tároljuk az elemeket.
- A kivételkor keressük meg a megfelelő prioritást, figyelembe véve a FIFO elvet is.

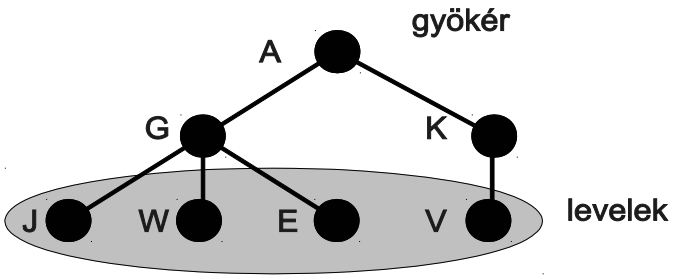
A prioritásos sor hatékony implementálására használhatjuk a későbbiekben bemutatott kupac tulajdonságot teljesítő bináris fát is.

Bináris fa adatszerkezet

A láncolt lista általában rugalmasabb eszköz adatok tárolásához, azonban lineáris/szekvenciális tulajdonsága miatt nehéz vele hierarchikus adatszerkezetet létrehozni. Habár a verem és a sor is valamilyen értelemben hierarchikus, azonban itt ez a hierarchia csak egy dimenziót takar. Az objektumok hierarchikus tárolására sokkal alkalmasabb a fa adatszerkezet.

A gráf egy olyan adatszerkezet, mely csúcsokat/csomópontokat és a csúcsok közötti éleket/összeköttetéseket tartalmaz. A fa egy olyan speciális gráf, mely bármely két csúcsát pontosan egy út köti össze, azaz a fa egy összefüggő és körmentes gráf.

A fában csúcsokat (*node*) és a csúcsok közötti kapcsolat (hierarchia) szervezésére éleket (*arc*) különböztetünk meg. A fa egy olyan hierarchikus adatszerkezet, melyben egy csúcsnak legfeljebb egy megelőzője/szülője (*parent*) lehet, azonban akárhány rákövetkezője/gyermeke lehet. Azokat a csúcsokat, melyeknek egyetlen gyermekük sincs levélnek (külső csúcs), azt a csúcsot, melynek nincs szülője gyökérnek nevezzük (*root*). Azokat a csúcsokat, melyek nem külső csúcsok, belső csúcsoknak nevezzük. Hagyományosan a fákat a hierarchiának megfelelően szintekbe szervezve ábrázoljuk. Egy szinten azok a csúcsok helyezkednek el, amelyek ugyanolyan távolságba vannak a gyökértől. Az ábrákon általában a gyökér csúcsot legfelül, a legfelső szinten szerepeltetjük. A 9. ábra megfelelően egy fát ábrázol.

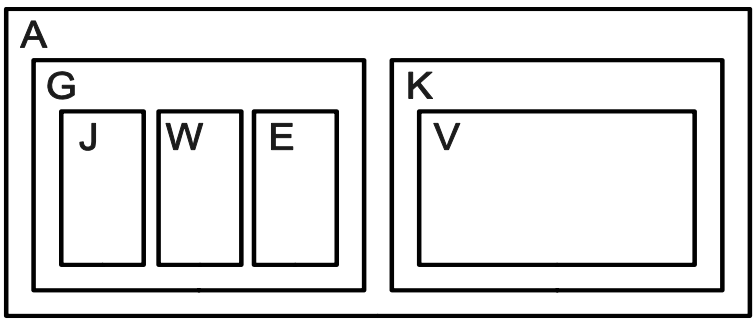


9. ábra: Fa adatszerkezet

Az előző ábrán látható fát ábrázolhatjuk Venn-diagrammal, vagy halmazokkal is. Az előző ábrán szereplő gráf halmazokkal ábrázolva:

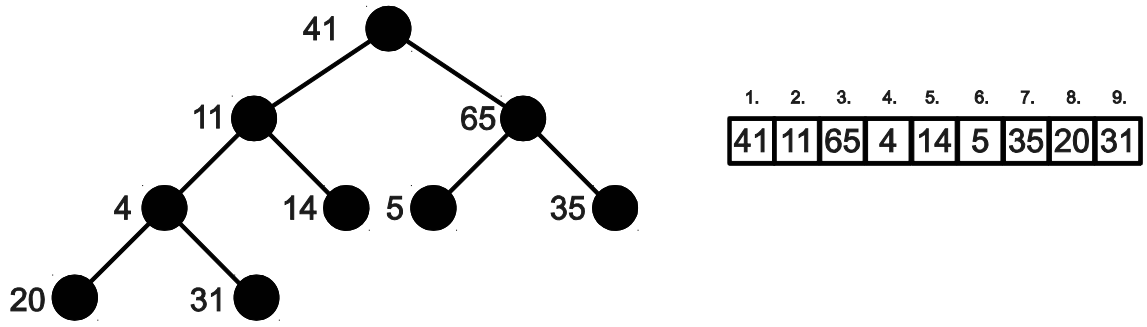
A={G, K}, G={J, W, E}, K={V}

A 9. ábrán látható fa Venn-diagrammal ábrázolva a 10. ábrán látható. A Venn-diagrammal ábrázolhatjuk a halmazok közötti összefüggéseket és így ezzel a fa adatszerkezetből eredő hierarchiát. Hasonlítsuk össze a kapott Venn-diagrammot a kezdetben vizsgált fa adatszerkezettel.



10. ábra: Fa ábrázolása Venn-diagrammal

A fa definíciója alapján egy csúcsnak akármennyi gyermeke lehet. A fákat osztályozhatjuk az alapján, hogy egy csúcsnak legfeljebb mennyi gyereke lehet. Azokat a fákat, melyeknél bármely csúcs legfeljebb két gyermeket tartalmaz bináris fáknak (*binary tree*) nevezzük. A bináris fa teljes (*full binary tree*), ha minden nem-levél csúcsnak pontosan két gyerek csúcsa van és minden levélből ugyanolyan hosszú úton érhető el a gyökér. A majdnem teljes bináris fa egy olyan fa, mely az utolsó szintjét kivéve teljesen kitöltött, azonban az utolsó szinten csak egy adott csúcsig vannak elemek (balról jobbra). Az n csúcsú, majdnem teljes bináris fa $O(\log n)$ szintet tartalmaz. Bináris fát többféleképpen is implementálhatunk. Egyik lehetőség, ha tömböt használunk a bináris fa elemeinek a tárolására, a másik lehetőség, ha egy láncolt lista segítségével tároljuk a fa adatait. A 11. ábrán egy majdnem teljes bináris fa és az azt ábrázoló tömb látható.



11. ábra: Majdnem teljes bináris fa hagyományos módon ábrázolva és tömbben tárolva

A fa ábrázolásához használt tömböt szintenként, fentről lefele haladva és szinten belül balról jobbra lépkedve töltjük fel. Mivel egy h szintet tartalmazó bináris fa legfeljebb 2^{h+1}-1 csúcsot tartalmaz, ezért előre tervezni tudjuk a bináris fa tárolásához szükséges tömb méretét. A tömbben az i. indexű elem gyerekei a $2i$. és a $2i$+1. indexű elemek, az i. indexű elem szülője pedig az $\lfloor i/2 \rfloor$. indexű elem. (Megj.: C és a C++ programozási nyelvben a tömb indexelése nullától indul, ezért fa tömbbel való ábrázolása esetén gondoskodjunk erről a tömb indexek kezelésekor.)

Bináris fa implementálása mutatókkal

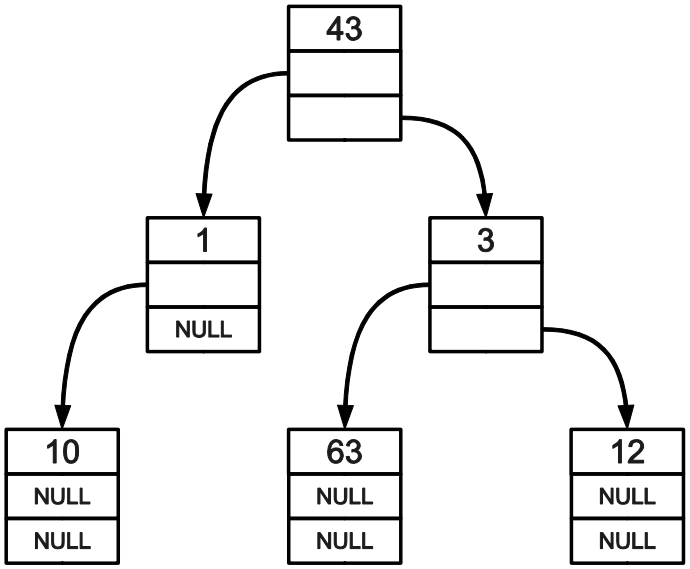
A majdnem teljes bináris fa tömbbel való tárolása helyett ebben a fejezetben olyan adatszerkezetet mutatunk be, mely mutatókat használ a fa csúcsok közötti kapcsolatok tárolására. A mutatókat használó adatszerkezet bármely bináris fa tárolására alkalmas, nem szükséges hozzá a majdnem teljesség tulajdonsága. Korábban láthattuk, hogyan lehet láncolt listába elemeket elhelyezni. A mutatókat használó fa adatszerkezet a láncolt listához hasonlóan biztosítja a csúcsok fába való szerveződését.

Bináris fa tárolásához egy fa csúcsnak ismernie kell a belőle elérhető jobb és a bal oldali részfát, amihez egy csúcsnak két olyan mutatót kell tartalmaznia, amelyek csúcs típusú objektumokra mutatnak. A következő mintafeladatban egy egész értékeket tároló bináris fa adatszerkezetet és a hozzá kapcsolódó metódusokat fogjuk létrehozni. Minden bináris fa csúcspont három adatmezőt tartalmaz: az adat tárolásához szükséges (data), a bal (left) és a jobb (right) oldali részfa mutatóit (megj. bizonyos feladatoknál, például bináris keresőfák, szokás még egy szülő pointert is tárolni a hatékonyság növelésének céljából).

```
struct IntTreeNode {
    int data;
    IntTreeNode *left;
    IntTreeNode *right;
```

```
}
```

A left és a right mutató abban az esetben, ha egy csúcsnak nincs bal, vagy jobb oldali részfája ezt NULL értékkel jelzi. A 12. ábrán egy mutatókkal szervezett bináris fa adatszerkezetet láthatunk. A fa gyökér eleme a 43 értéket tárolja, a fa három levél csúcsot tartalmaz.



12. ábra: Bináris fa mutatók segítségével ábrázolva

Bináris fa csúcsainak megszámlálása

A bináris fa bármely csúcsából elérhető részfa szintén egy bináris fát alkot. Ennek a rekurzív tulajdonságnak a kihasználásával könnyen megvalósíthatunk egy olyan rekurzív eljárást, mely a bináris fa csúcsait számolja meg. Ezt az eljárást a countIntNodes() függvény segítségével implementáltuk.

```
int countIntNodes(IntTreeNode *root) {
    if ( root == NULL )
        return 0;
    else {
        int count = 1;
        count += countIntNodes(root->left);
        count += countIntNodes(root->right);
        return count;
    }
}
```

A csúcsokat megszámláló algoritmust legegyszerűbb rekurzív módon szervezni. A rekurzív megállási feltételt a mutató NULL értéke biztosítja. Amennyiben a részfára mutató érték nem NULL, akkor össze kell adni a jobb oldali és a bal oldali részfában szereplő csúcsok számát.

Az algoritmust rekurzió nélkül is meg lehet valósítani. A probléma a fában bejárt út tárolásából és nyomkövetéséből adódik. A rekurzív függvény hívások során egy verem adatszerkezetbe kerülnek a hívó függvények. A rekurzív hívásokat egy saját verem adatszerkezettel is helyettesíthetnénk.

Bináris fa bejárása

A fa bejárás egy olyan művelet, mely során a fa minden csúcsát pontosan egyszer látogatjuk meg. A fa bejárása felfogható úgy is, hogy valamely módszernek megfelelően egymás után tesszük a fa csúcsait, vagyis linearizáljuk a fát. A bejárástól függően más-más sorrendben rakhatjuk sorba a fa csúcsait. Mivel n darab különböző csúcsnak $n!$ sorrendje lehetséges, ezért $n!$ különböző bejárás létezik. Ezeknek a nagy része gyakorlati értelemben nincs jelentősége, de vannak olyan fa bejárások, melyek bizonyos alkalmazásokban hasznunkra válhat. Ezek közül a legfontosabb kettő bejárást, a szélességi és a mélységi fa bejárást részletesen áttekintjük.

Szélességi bejárás

Szélességi (szintfolytonos, BFS, breadth-first search/traversal) bejárás esetén a gyökértől elindulva haladunk lefele a szinteken keresztül. Egy csúcsot akkor „járhatunk be”, ha a fölötte levő szintek csúcsain már jártunk. A feladatot egy iteratív algoritmussal oldhatjuk meg. A soron következő csúcsok tárolására egy IntTreeNode* mutatókat tartalmazó sort (queue) használhatunk. A sor kezdetben a gyökér csúcsot tartalmazza. Minden egyes iterációban kivesszük a sor első elemét és a kivett elem gyermekeit berakjuk a sor végére.

```
void BFSIntTree(IntTreeNode *root) {
    IntTreeNodeQueue queue;
    IntTreeNode *p;
    if(root != 0)
        queue.enqueue(root);
    while(!queue.isEmpty()) {
        p = queue.dequeue();
        cout << p->data << " "; //visit p
        if(p->left != 0)
            queue.enqueue(p->left);
        if(p->right != 0)
            queue.enqueue(p->right);
    }
}
```

A 12. ábrán látható bináris fára a szélességi bejárás a következő sorrendben írja ki a fa csúcsaiban tárolt értékeket: 43, 1, 3, 10, 63 és 12.

Mélységi bejárás

Mélységi bejárás (DFS, *depth-first search/traversal*) esetén a gyökértől indulva haladunk olyan „mélyre” a szinteken, ameddig csak lehet. Ha már nem lehet mélyebbre haladni, mert levélhez

értünk, vagy pedig minden elérhető csúcs meglátogatásra került, akkor visszalépünk az előző szintre és erről a szintről próbálunk másik csúcsok irányába továbbhaladni lefelé.

Mélységi bejárás megvalósítható egy rekurzív függvénnyel. A rekurzív függvény például megvalósítható úgy, hogy addig, amíg a bal oldali részfája létezik egy csúcsnak haladjunk a bal oldali részfán keresztül lefele. Ha nem tud a baloldali részfákban továbbhaladni, akkor visszalépés és a jobb oldali részfák felderítése történik.

A mélységi bejárás többféleképpen is megvalósítható attól függően, hogy a jobb-, baloldali részfa és az aktuális csúcs feldolgozásának mi a sorrendje. Három olyan sorrendet különböztetünk meg, ami fontos szerepet tölt be informatikai algoritmusokban:

- Preorder bejárás: aktuális csúcs – baloldal – jobboldal
- Inorder bejárás: baloldal – aktuális csúcs – jobboldal
- Postorder bejárás: baloldal – jobboldal – aktuális csúcs

A preorder, inorder és postorder fa bejárás rekurzív függvényekkel a következőképpen implementálható:

```
void Preorder(IntTreeNode *p) {
    if(p!=0) {
        cout << p->data;
        Preorder(p->left);
    Preorder(p->right);
    }
}
void Inorder(IntTreeNode *p) {
    if(p!=0) {
        Inorder(p->left);
        cout << p->data;
        Inorder(p->right);
    }
}
void Postorder(IntTreeNode *p) {
    if(p!=0) {
        Postorder(p->left);
        Postorder(p->right);
        cout << p->data;
    }
}
```

A 12. ábra bináris fája a különböző típusú mélységi bejárások esetén a következő sorrendben írja ki a fában tárolt értékeket:

- Preorder: 43, 1, 10, 3, 63, 12

- Inorder: 10, 1, 43, 63, 3, 12
- Postorder: 10, 1, 63, 12, 3, 43

A Preorder, Inorder és Postorder fa bejárások természetesen megvalósíthatóak rekurzió nélkül is oly módon, hogy a rekurzív függvényhívásokat egy veremmel helyettesítjük. Mindhárom bemutatott bejárás esetén minden fa csúcs pontosan egyszer kerül be a tárolóba, ezért az algoritmusok komplexitása O(n), ahol n a fa csúcsainak száma.

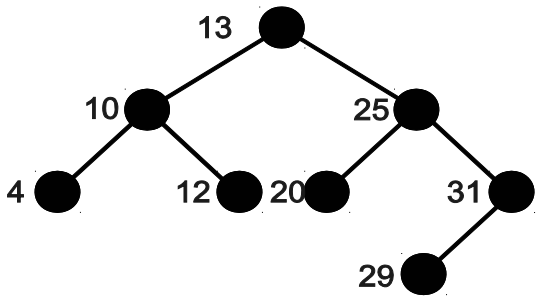
Bináris keresőfa

Egy adott érték tömbben való megkeresése átlagosan a tömbméret felével megegyező lépéssel oldható meg. Ha a tömb elemeket rendezve tároljuk, akkor a keresés ennél még hatékonyabban oldhatjuk meg a bináris keresés módszerével. Bináris (vagy logaritmikus) keresés során például a tömb középső elemének megvizsgálásával el tudjuk dönteni, hogy a rendezett tömb elején, vagy a végén található a keresett érték. Egy-egy ilyen összehasonlítással a tömb aktuális elemeinek a felét kizárhatjuk a keresésből, egészen addig míg a kizárások után a vizsgálatban marad tömbelemek száma egy, vagy kettő lesz.

A keresés a rendezett tömbben a bináris keresés módszerével $O(\log n)$ idő alatt végrehajtható. Ugyanakkor a rendezett tömbbe való új elem beszúrás ugyanolyan erőforrás igényes művelet mint a nem rendezett tömb esetén. Azért hogy a tömb rendezett maradjon a beszúrás után is meg kell keresni az új elem helyét, és ettől a helytől kezdődően a tömb végén levő tömbelemeket eggyel el kell mozgatni hátrafelé, hogy biztosítsuk a helyet az új tömbelem számára. Hasonlóan a rendezett tömbből való elem törlése is a tömb megmaradó elemeinek mozgatását eredményezi. Ezeket az elemek mozgatásával járó műveleteket kerülhetjük el, ha a rendezett tömb helyett rendezett listában tároljuk az értékeket. Ugyanakkor a rendezett láncolt listában nehéz az elemek közvetlen címzése (például a 16. pozícióban való elem megkereséséhez a lista fejétől indulva végig kell lépkedni a 16. pozíció előtt szereplő összes listaelemen).

Mind láncolt listában, mind tömbben lehet rendezett módon adatot tárolni (pl. növekvő sorrendben). Egy rendezett tömb vagy lista esetén egy új adat hozzáadásakor meg kell keresni a rendezésnek megfelelő pozícióját az új adatnak, majd ebbe a pozícióba be kell láncolni, vagy a tömbbe beszúrni. A megfelelő pozíció megkereséséhez mindkét adatstruktúra esetén az elemek végignézését és összehasonlítását kell tennünk. Egy bizonyos adatmennyiségig a láncolt listás és a tömbös megoldás is jól használható. Az adatok rendezetten való tárolására egy sokkal hatékonyabb eszköz, ha bináris keresőfát használunk. A bináris kereső fa tömb és a láncolt listában való rendezett módon való tárolás előnyös tulajdonságait egyesíti egy adatszerkezetbe.

A bináris keresőfa egy bináris fa, melyre teljesül a bináris keresőfa tulajdonság. A bináris keresőfa tulajdonság szerint bármely csúcs esetén teljesülnie kell, hogy a csúcs bal oldali részfájában a csúcs értékénél kisebb, a jobb oldali részfájában pedig csak nagyobb vagy egyenlő értékek szerepelhetnek. A 13. ábrán bináris kereső tulajdonságot teljesítő fa látható.



13. ábra: Bináris kereső tulajdonságot teljesítő fa

A bináris keresőfa tárolására használhatjuk a bináris fák tárolására bemutatott adatszerkezeteket. Ha van egy bináris keresőfánk, akkor abban megvalósíthatjuk a keresés műveletet. A keresés művelet során a bináris keresőfa tulajdonságot kell csak sorozatosan kihasználni. Ha a keresett érték megegyezik az aktuális csúcs értékével, akkor megtaláltuk, amit kerestünk, ha kisebb nála, akkor a bal oldali, ha nagyobb nála, akkor a jobb oldali részfában kell rekurzív módon tovább keresni.

A következő példában egy osztályt hozunk létre bináris kereső fa megvalósítására. A bináris keresőfához a korábban ismertetett bináris fa adatszerkezetet használjuk. Ebben a bináris fa adatszerkezetben egész értékek bináris fában való tárolására van lehetőségünk. A fa csúcsainak tárolására a következő osztályt használjuk:

```
class BSTIntNode{
public:
    BSTIntNode() {
        left = right = 0;
    }
    int key;
    BSTIntNode *left, *right;
}
```

A bináris keresőfához a BSTIntNode osztályt használjuk a bináris fa felépítésére. A bináris keresőfát a BST osztály segítségével implementáltuk. A BST osztályban a root adattag mutatja a bináris keresőfa gyökerét. A konstruktorban szerepe az adattagok inicializálása, a destruktor feladata pedig a felépített bináris kereső fa lebontása, a lefoglalt csúcsok felszabadítása. Az osztályban lehetőséget kell biztosítani a bináris kereső fa építésére és módosítására, az ezekhez szükséges függvényeket jelenleg nem implementáljuk.

```
class BST {
public:
    BST() { root = 0; }
    ~BST() { … }
    bool isEmpty() const { return root==0; }
    int searchTree(const int element) {
        return search(root, element);
    }
protected:
    int search(BSTIntNode*, const int) const;
private:
    BSTIntNode *root;
}
```

A BST osztályban a searchTree metódust használjuk arra, hogy egy adott érték szereplését ellenőrizzük a bináris keresőfában. A keresőfában minden esetben a gyökérelemtől indítjuk a keresést. A keresés végrehajtásához egy belső protected metódust használunk, mely iteratív módon ellenőrzi, hogy a keresett elem szerepel-e a fában? Az algoritmus minden egyes iterációban egy összehasonlítást végez: a keresett érték megegyezik-e, az aktuális csúcsban tárolt értékkel, kisebb, vagy nagyobb-e nála. Az összehasonlítás eredményétől függően megtalálja az értéket, vagy eldönti, hogy melyik részfában kell tovább keresnie (a másik részfa kizárható a továbbiakban a keresésből). A search metódust a következőképpen implementálhatjuk.

```
int BST::search(BSTIntNode *p, const int element) const {
    while(p != 0) {
        if(element == p->key)
            return p->key;
        else if(element < p->key)
            p = p->left;
        else
            p = p->right;
    }
    return 0;
}
```

Keresés során legrosszabb esetben, amikor a keresett érték nem található meg a bináris kereső fában, vagy levél szinten van meg benne, akkor a fa magassága számú iterációt kell végrehajtani. Ha az érték valamely magasabb szinten található, akkor ennél kevesebb összehasonlítással megtaláljuk a keresett elemet.

Bináris kereső fa esetén inorder fa bejárással visszakapjuk a fában tárolt értékeket növekvő sorrendben. A bináris kereső fa tulajdonság biztosítja, hogy inorder bejárás esetén minden bal oldali részfában szereplő csúcs előbb kerül kiírásra, mint az aktuális csúcs és a jobb oldali részfában szereplő csúcsok pedig később.

Ha egy bináris keresőfába új elemet szeretnénk berakni, akkor meg kell keresni azt az első olyan szabad pozíciót, melybe ha beszúrjuk az új elemet a bináris kereső fa tulajdonság nem sérül. Ennek a pozíciónak a megkereséséhez használhatjuk a search függvényben bemutatott iterációt. Ha megtaláltuk azt a helyet, ahova beláncolhatjuk az új értéket, akkor egy új csúcs lefoglalásával és a bináris kereső fához való csatolásával elvégezhetjük a beszúrás műveletet.

A törlés művelet esetén több esetet kell megvizsgálni. Amennyiben a törlendő elemnek nincs gyermeke, akkor csak a szülő megfelelő (left/right) mutatóját. Ha a törlendő elemnek pontosan egy gyermeke van, akkor az elem törlése megvalósítható, ha egyszerűen „kiláncoljuk” a fából, azaz a szülőjének a megfelelő (left/right) mutatóját a törlendő elem gyermekére irányítjuk. Ha a törlendő elemnek két gyermeke is van, akkor azt a legközelebbi rákövetkező elemmel cseréljük ki a benne tárolt értéket, melynek nincsen bal oldali részfája. A kicserélés után a gyermek nélküli, vagy az egy gyermekes elem törlésének szabályai alapján végezzük el az elem törlését.

A keresés és a beszúrás akkor maradhatnak hatékony műveletek a bináris keresőfában, ha a fa kiegyensúlyozott (*balanced*) marad. Akkor kiegyensúlyozott egy bináris kereső fa, ha a gyökérből a levelekbe vezető utak közel azonos hosszúságúak, azaz minél „közelebb” van a majdnem teljes bináris fákhoz. Más szavakkal bármely csúcs esetén a jobb és a bal oldali bináris részfában közel azonos mennyiségű csúcs szerepelhet. A tökéletesen kiegyensúlyozott bináris

kereső fában a részfákban szereplő csúcsok számának különbsége legfeljebb egy. Ilyen tökéletesen kiegyensúlyozott fát nehéz folyamatosan karbantartani, azonban egy véletlenszerűen bővülő és csökkenő bináris keresőfa esetén egy nagyjából kiegyensúlyozott bináris keresőfát kapunk. A kiegyensúlyozott bináris keresőfában egy keresési lépésben az egyik részfát kizárjuk a keresésből, ezzel körülbelül a felére csökkentjük azoknak a csúcsoknak a számát, amiket figyelembe kell venni a továbbiakban.

Bináris kereső fában a legkisebb elemet megkaphatjuk a gyökér elemtől a bal oldali részfákban levél szintek felé lépegetve.

```
int BST::minimum(BSTIntNode *p) const {
    while(p->left != 0) {
        p = p->left;
    }
    return p->key;
}
```

Hasonlóan a legkisebb elem kereséséhez a legnagyobb elemet megkaphatjuk a jobb oldali részfákban a levél szint felé haladva.

```
int BST::maximum(BSTIntNode *p) const {
    while(p->right != 0) {
        p = p->right;
    }
    return p->key;
}
```

Piros-fekete és AVL fák

A piros-fekete fa olyan speciális bináris kereső fa, melyben minden csúcshoz hozzárendelünk egy színkódot (piros/fekete). A színezésre korlátozások bevezetésével biztosítható, hogy a fa kiegyensúlyozott maradjon oly módon, hogy a fában a gyökérből induló leghosszabb út hossza nem lehet nagyobb, mint a gyökérből induló legrövidebb út hosszának a kétszerese.

Az AVL fa olyan bináris fa, mely magassága kiegyensúlyozott, azaz bármely fa csúcsra teljesül, hogy a csúcs bal oldali részfájának és a jobb oldali részfájának a magassága legfeljebb 1-el különbözik.

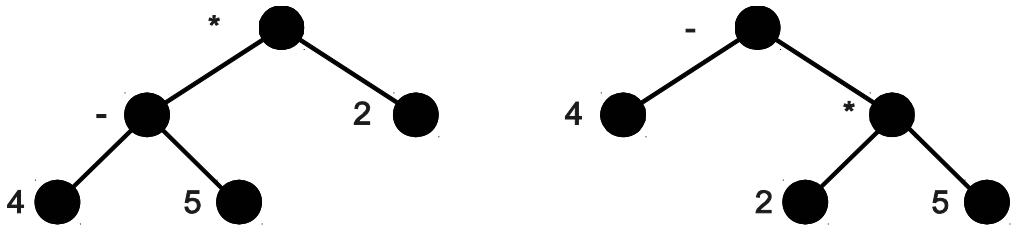
Kifejezések tárolása bináris fával

Bináris fák egyik fontos alkalmazása aritmetikai és logikai kifejezések tárolása és kiértékelése lehet. Ennél az alkalmazásnál elengedhetetlen, hogy a tárolás során egyértelmű legyen a kifejezés és a bináris fa kapcsolata azért, hogy a kifejezés kiértékelésével egyértelműen a matematikailag helyes megoldást kapjuk.

Az 1920-s években egy lengyel matematikus, Jan Lukasiewicz vezetett be egy olyan speciális, zárójeleket nem tartalmazó formát az aritmetikai kifejezések ábrázolására, amelyből a műveletek elvégzésének sorrendje egyértelműen adódik. Ezt az ábrázolást lengyel formának nevezzük. Habár a lengyel forma kevésbé olvasható ábrázolás, mint a zárójeleket használó

hagyományosan elterjedt ábrázolás, de a számítástechnikában fordítóknak és értelmezőknek ez a formula könnyebben kiértékelhető és kezelhető ábrázolást ad.

Vegyük például a 4-5*2 aritmetikai kifejezést. A kifejezés értéke függ a kiértékelés sorrendjétől, amit a zárójelezéssel irányíthatunk: (4-5)*2 = -2, ugyanakkor 4-(5*2) = -6. A kifejezés lengyel formájából könnyen építhető egy olyan bináris fa, mely tartalmazza a kiértékeléskor követett sorrendet (azaz a zárójelezést és a műveleti jelek precedenciáját). A 14. ábrán két bináris fát láthatunk, amelyek a két különböző zárójelezést jelenítik meg. A bináris fák egyértelműen ábrázolják az adott kifejezést. Levél szinten találhatóak a számok, köztes csúcsokban pedig a műveletek. Ahhoz, hogy egy csúcs értékét megkapjuk ki kell értékelni előbb a jobb és a bal oldali részfát, majd utána tudjuk a kapott eredményekkel a csúcsban szereplő műveletet elvégezni.



14. ábra: Aritmetikai kifejezés ábrázolása bináris fával

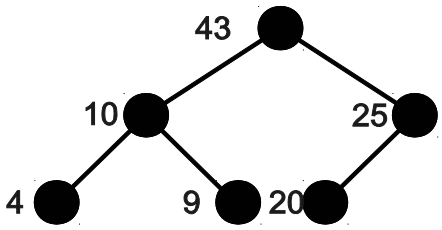
Vegyük észre, hogy a fa nem tartalmazza a zárójeleket, ugyanakkor egyértelműen leírható vele az aritmetikai kifejezés. A korábban ismertetett bináris fa bejárására bevezetett inorder, preorder és postorder fa bejárásokkal egy-egy kifejezést kapunk. A bal oldali bináris fa esetén a preorder bejárás a * - 4 5 2 sorrendhez, az inorder a 4 - 5 * 2, míg a postorder a 4 5 - 2 * sorrendhez vezet. A második fánál a preorder a - 4 * 2 5, az inorder a 4 - 2 * 5, a postorder pedig a 4 2 5 * - sorrendhez vezet. Ezek közül a bináris fa bejárások közül az inorder nem alkalmas, hogy egyértelműen leírja az elvégzendő matematikai műveletek sorrendjét, hiszen mindkét fa esetén ugyanazt a kifejezést generálja. A másik két bejárás azonban alkalmas az egyértelmű ábrázolásra majd a feldolgozásra. Jelentőségük miatt a preorder bejárással kapott kifejezést prefix, a postorder bejárással kapott kifejezést pedig postfix kifejezésnek nevezzük. A prefix ábrázolásban a művelet megelőzi az operandusokat, a postfixben az operandusok előbb szerepelnek, mint maga a művelet. Vannak olyan programozási nyelvek, melyek prefix (LISP) és vannak olyanok, melyek postfix (LOGO) jelölést használnak. Mind a kifejezést lengyel formára alakító, mind a lengyel formából a kifejezés értékét kiszámító algoritmus egy-egy szép példája a verem alkalmazásának.

Kupac adatszerkezet

A kupac (*heap*) egy olyan speciális bináris fa, melyre teljesül az, hogy bármely csúcs értéke nem kisebb mint a csúcs bármely gyerekének az értéke, továbbá a fa majdnem teljes, azaz a legalsó szint kivételével minden szinten teljesen kitöltött, a legalsó szinten balról jobbra haladva egy adott csúcsig az összes elem megtalálható. A továbbiakban az ilyen bináris fákat kupac tulajdonságot teljesítő bináris fának nevezzük. Tudjuk, hogy az n elemet tartalmazó majdnem teljes bináris fa magassága legfeljebb $O(\log n)$.

A majdnem teljes bináris fa tárolására használhatjuk a tömb adatszerkezetet (lásd 11. ábra). A kupac tulajdonságai miatt a tömb legnagyobb eleme a fa gyökéreleme, amit a tömb első indexén tárolunk. Kupac segítségével gyorsan vissza tudjuk adni a legnagyobb elemet, azonban mivel a többi elem nem rendezett, ezért ugyanazokhoz az elemekhez több különböző kupac is építhető. A 15. ábrán egy kupacot láthatunk. Kupac tulajdonságot és a kupacot a későbbiekben a kupacrendezés algoritmusában fogjuk használni. A kupac a rendezés mellett a prioritásos sorok létrehozásában kap fontos szerepet. Hasonlóan építhető olyan kupac is, melynek tetején

(gyökerében) a legkisebb elem található, így beszélhetünk maximum és minimum kupacról, aszerint, hogy a maximális, vagy minimális elem kerül a kupac tetejére.



15. ábra: Kupac tulajdonságot teljesítő bináris fa

Rendezés

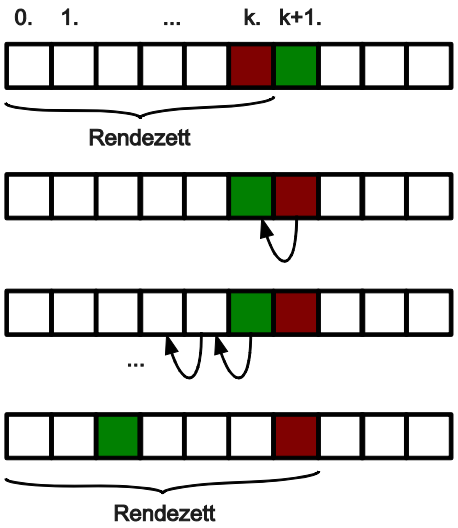
Adatok hatékony kezeléséhez elengedhetetlen az adatokat valamilyen szempont alapján rendezve tárolni. Mindennapi példák mutatják, hogy meg lehet találni valamit rendezetlen adatok között is, azonban a keresést felgyorsíthatja, ha az adatok rendezve vannak (pl. könyvtár, telefonkönyv). Kis adatmennyiség esetén (10-100-1000 darab) mindegy, hogy mennyire gyors, mennyire hatékony az algoritmus, mely a rendezést végzi. Ha azonban az adatmennyiség növekszik, a rendező algoritmus hatékonysága nagyban befolyásolhatja az egész alkalmazás használhatóságát.

A rendezés első lépése a rendezési elv meghatározása. A rendezési elv kiválasztását általában az alkalmazás határozza meg. Számok esetén a rendezési elv lehet a növekvő, vagy a csökkenő sorrend. Nevek esetén általában ábécé szerinti sorrendet használunk.

A következő rendező algoritmikusok esetén tömbben tároljuk azokat az adatokat, melyekre az algoritmusok működését bemutatjuk. Az algoritmikusok összehasonlításon alapuló helyben rendező algoritmusok, azaz a tömb elemek egymással való összehasonlítása és értékek kicserélésével határozzák meg a rendezett tömböt. Feltételezzük, hogy az adatok egész számok és a rendezési elv a növekvő sorrend.

Beszúrásos rendezés

A beszúrásos rendezés a tömb elején egy rendezett résztömb lépésenkénti bővítésével határozza meg a teljes rendezett tömböt. A beszúrásos rendezés (*insertion sort*) első lépéseben a tömb első két elemét hasonlítja össze (d[0] és d[1]), a tömb második elemét szúrja be az egy hosszúságú rendezett résztömbbe. Ekkor az első két eleme egymáshoz képest rendezve van. Következő lépésben a harmadik elemmel (d[2]) kibővítve rendezi az első két elemet. Az algoritmus megkeresi a d[2] helyét a d[0] és d[1] elemekhez képest, beszúrja a d[2]-t a helyére, a többi elem elcsúsztatásával készíti elő a d[2] helyét. A k. lépésben az első k elem már rendezve van. Az algoritmus azokat az elemeket, melyek nagyobbak mint a d[k] elem eltolja egy hellyel jobbra, így készít helyet a d[k] tömbelem számára. Az algoritmus lépéseinek és működésének a szemléltetését a 16. ábrán láthatjuk.



16. ábra: Beszúrásos rendezés működésének szemléltetése

A beszúrásos rendezés egybeágyazott for ciklusok segítségével implementálható. A for ciklusok egybeágyazása jelzi az algoritmus komplexitását ($O(n^2)$). A beszúrásos rendezés C++ forráskódja a következőképpen épülhet fel:

```
insertionSort(int d[], int size){
    int tmp;
    for(int i=1, j;i<size;i++){
        tmp=d[i];
        for(j=i;j<0 && tmp<d[j-1];j--)
            d[j]=d[j-1];
        d[j]=tmp;
    }
}
```

Az insertionSort függvény a size méretű d tömböt rendezi. A külső for ciklus egyre növeli a d tömb első felében levő rendezett résztömb méretét. A belső for ciklus az utoljára a résztömbhöz hozzáadott elem helyét keresi meg oly módon, ha egy előtte szereplő tömbelem értéke nagyobb, akkor azt egymás után eggyel-eggyel jobbra tolja el. Az eltolás közben végrehajtott érték cserékhez a tmp segédváltozót használja.

A következő példában az algoritmus működését szemléltetjük. Legyen a d tömb mérete 5, a d tömb elemei pedig legyenek 4,1,9,2,8 egészek. A következő négy táblázat a d tömb értékeinek változását ábrázolja külső for ciklus közben. A narancs háttérszín azt a résztömböt jelöli, mely már rendezve van. Az i=2 iterációban nem történik változtatás, hiszen a rendezett résztömbhöz olyan új elemet adtunk hozzá, mellyel továbbra is rendezett maradt a tömb. Az i=3 iterációban két lépésben találta meg az új elem helyét a rendezett résztömbben.

i=1, tmp=1	0	1	2	3	4
d ciklus előtt	4	1	9	2	8
d ciklus közben		4	9	2	8
d ciklus után	1	4	9	2	8

i=2, tmp=9	0	1	2	3	4
d ciklus előtt	1	4	9	2	8
d ciklus után	1	4	9	2	8

i=3, tmp=2	0	1	2	3	4
d ciklus előtt	1	4	9	2	8

d ciklus közben	1	4		9	8
d ciklus közben	1		4	9	8
d ciklus után	1	2	4	9	8

i=4, tmp=8	0	1	2	3	4
d ciklus előtt	1	2	4	9	8
d ciklus közben	1	2	4		9
d ciklus után	1	2	4	8	9

Kiválasztásos rendezés

A kiválasztásos rendezés (*selection sort*) során az adatok másolásának mennyiségét igyekszik csökkenteni oly módon, hogy az algoritmus keres egy olyan elemet, ami nem a helyén található, majd rögtön a végleges helyére másolja. Első lépésben megkeresi a legkisebb tömbelemet, amit a tömb első helyén szereplő elemmel cserél ki. Innentől kezdve az első elemmel már nem kell foglalkozni, elég a második elemmel kezdődő tömböt rendezni.

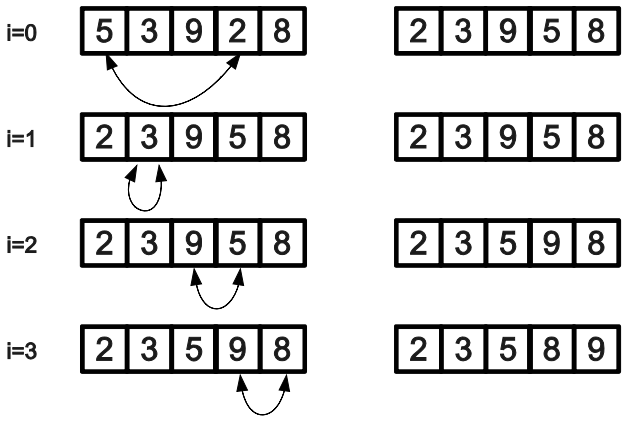
A beszúrásos rendezéshez hasonlóan a kiválasztásos rendezés is négyzetes jellegű algoritmus ($O(n^2)$). Az algoritmus egybeágyazott for ciklusok segítségével implementálható. A kiválasztásos rendezés C++ forráskódja a következő:

```
selectionSort(int d[], int size){
    int tmp,least;
    for(int i=0, j;i<size-1;i++){
        for(j=i+1, least=i;j<size;j++)
            if(d[j]<d[least])
        least=j;
        tmp=d[i];
        d[i]=d[least];
        d[least]=tmp;
    }
}
```

A selectionSort algoritmus a size elemet tartalmazó d tömböt rendezi. A tmp segédváltozóra az adatok cseréjéhez, a least változóra az aktuálisan legkisebb tömbelem indexének tárolására van szükség. A külső for ciklusban futó i index azt jelöli, hogy honnét kezdődik az a résztömb, melyben a legkisebb elemet kell keresni. Ennek a ciklusnak elég a size-1 elemig futnia, mert az utolsó lépésben egy 1 elemű tömb már mindenféleképpen rendezettnek tekinthető. A belső

ciklus az i. elemtől indulva keresi meg az i. elem és a tömb utolsó eleme közül a legkisebb elem indexét. A ciklus befejeztével a legkisebb elem és az i. elem cseréjét végezzük.

Az algoritmus viselkedésének szemléltetésére rendezzük az 5 elemből álló 5, 3, 9, 2, 8 tömböt a selectionSort algoritmussal. A 17. ábrán láthatjuk az algoritmus futása során a d tömb változását. Az i=1 iterációban a 2. és 5. elem közül a legkisebbet kell megkeresni és kicserélni a 2. helyen álló elemmel. Mivel ebben a példában a legkisebb elem éppen a 2. pozícióban található, ezért önmagával kellene kicserélni. A selectionSort algoritmus ezekre az esetekre kiegészíthető egy ellenőrzéssel, ami alapján csak akkor hajtsuk végre a cserét, ha least változó nem egyenlő az i változóval (enélkül a kiegészítés nélkül is helyesen rendezi a tömböt az algoritmus).



17. ábra: A selectionSort algoritmus lépéseinek bemutatása

Buborék rendezés

A buborék rendezés (*bubble sort*) bemutatásához a rendezendő tömböt ábrázoljuk vertikálisan, egy oszlopban. Az algoritmus a tömb aljától felfele halad. Megvizsgálja az egymás mellett álló elemeket és ha nem jó a sorrendjük, akkor megcseréli őket. Egy size méretű tömb esetén először a d[size-1] és d[size-2] elemeket hasonlítja össze és cseréli ki őket, ha szükséges. Következő összehasonlítás és az esetleges csere a d[size-2] és d[size-3] elemek között történik. Az összehasonlítások és cserék egészen a tömb „tetejéig” a d[1] és d[0] elemig történik. Ezzel, a tömb egyszeri végignézésével a tömb tetejére „felbuborékoltattuk” a tömb legkönnyebb (legkisebb) elemét.

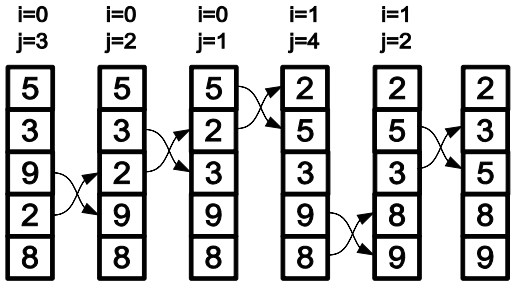
A következő lépésben ugyanezt a módszert hajtjuk végre, azonban most elég a tömb utolsó előtti eleméig haladni felfele, hiszen a legkisebb elem már a helyén van (d[0]). A tömb második végigjárásával a két legkisebb tömbelem a 0. és az 1. pozícióban található. Az algoritmus addig folytatódik, ameddig a tömb végigjárás a legalsó elemek összehasonlítására és cseréjére redukálódik.

A buborék rendezés, hasonlóan a korábbi rendezési algoritmusok, is négyzetes jellegű algoritmus (komplexitása $O(n^2)$). Az algoritmus egybeágyazott for ciklusok segítségével implementálható. A buborék rendezés C++ forráskódja a következő:

```
bubbleSort(int d[], int size){
    int tmp;
    for(int i=0;i<size-1;i++){
        for(j=size-1;j>i;--j)
            if(d[j]<d[j-1]){
```

```
            tmp=d[j];
            d[j]=d[j-1];
            d[j-1]=tmp;
        }
    }
```

Rendezzük a 5,3,9,2,8 egészeket tartalmazó tömböt a buborékrendező algoritmus segítségével. A 18. ábrán az algoritmus végrehajtása során azokat a lépéseket ábrázoljuk, melyek során a tömbelemek cseréje történt.



18. ábra: Buborék rendezés lépései az 5,3,9,2,8 elemeket tartalmazó tömbre

Kupac rendezés

A kupac rendezés (*heap sort*) a kiválasztásos rendezés működését tekintve nagyon hasonlít rá. Mindkét algoritmus kiválasztja az aktuálisan legkisebb, vagy legnagyobb elemet a még rendezetlen elemek közül és berakja azt a rendezésnek megfelelő helyre. Ezt a kiválasztást addig folytatják, míg a tömb elemei rendezettek lesznek.

A kupac rendezés a kupac tulajdonságot kihasználva rendezi a tömböt. Ha növekvő sorrendbe akarjuk rendezni a tömböt, akkor a kupacból a legnagyobb elemet a tömb legutolsó helyére kellene tenni. A kupac tulajdonság alapján a legnagyobb érték a bináris fa gyökéréhez tartozik, ami a fa tömbbel való tárolása esetén a tömb legelső elemének felel meg. Az algoritmusban a kupac gyökér elemét kicseréljük a kupac (vagyis a tömb) legutolsó elemével. A tömb utolsó eleme ezzel már a legnagyobb érték, ami már a helyére került, ezért „leválaszthatjuk” a kupacról (a kupac egyre kisebbre zsugorodik a tömbön belül). Ha a maradék elemeket kupacba rendezzük, folytathatjuk az algoritmust. Az algoritmus elindításához legelső lépésben egy tömbből kupacot kell készítenünk. A kupac rendezés a következő lépéseket tartalmazza:

```
heapSort(int d[], int size) {
    kupac készítése a d tömbből
    for(i=n-1;i>1;i--) {
        gyökér elem és az i elem cseréje
        kupac helyreállítása a d tömb 0..i-1 elemeiből
    }
}
```

Ahhoz, hogy a kupac rendezés működhessen, egy tetszőleges tömbből kupacot kell tudnunk létrehozni. Ehhez a tömbben levő elemeket újra kell szervezni oly módon, hogy a kupac tulajdonságot teljesítsük.

Többfajta algoritmus létezik kupac létrehozására. A legegyszerűbb, ha egy üres kupaccal indítunk és ehhez adunk hozzá új elemeket úgy, hogy továbbra is kupac maradjon (top-down). Ekkor a kupac végére teszünk be egy új elemet. Ha az új elem nem nagyobb, mint szülő értéke, akkor a kapott fa kupac, ha nagyobb, akkor a szülő elemmel kicseréljük. Ha cserélni kellett, akkor ezt a vizsgálatot végezzük fölfele a gyökér fele haladva egészen addig, míg nem kell cserélni, vagy elértünk a gyökérhez.

Meglevő tömb kupaccá alakítását egy új tömbbe való másolás helyett a lentről felfele építkezéssel végezhetjük el. Ebben az esetben a levelek irányából elindulva haladva hozunk létre kupacokat. Ebben az algoritmusban amikor egy csúcsból készítünk kupacot már a jobb és bal oldali részfáira teljesülnek a kupac tulajdonsága. Az algoritmus rekurzív, megnézzük hogy az aktuális csúcshoz, a jobb, vagy a baloldali gyermekhez tartozik a legnagyobb érték. Ha ez az érték nem az aktuális csúcs, akkor a legnagyobb értékkel kicseréljük az aktuális csúcsot, majd elindítjuk az algoritmust arra a részfára, ahol a csere történt. Ezt az algoritmust a heapify függvénnyel hozzuk létre.

Ha van egy ilyen algoritmusunk, ami kupaccá alakít egy csúcsból elérhető fát az esetben, ha a jobb és bal oldali részfa mindegyike kupac, akkor a tömbben hátulról előre fele haladva, az első olyan csúcsból indulva, mely már nem levél kupacot hozhatunk létre. Elég a nem levelektől indulni, hiszen az egy csúcsú fákra teljesül a kupac tulajdonság. Ezt az algoritmust a buildHeap függvényben implementáljuk.

A buildHeap függvény paraméterei, a bináris fát leíró d tömb és a tömb/kupac elemszáma. Az első nem levél csúcs tömbindexe bináris majdnem teljes fa és C++ tömbindexelés esetén (heapsize-1)/2. Minden iterációban elindítunk egy kupacba rendező heapify függvényt a tömb, az aktuális csúcs indexe és kupac méretének paramétereivel.

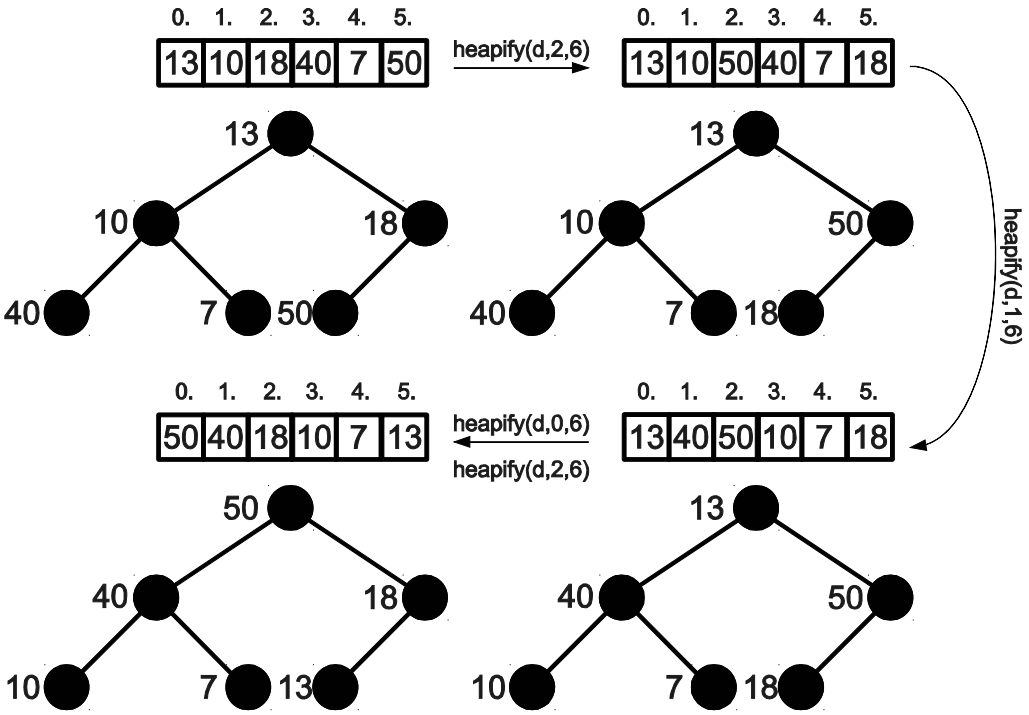
```
void buildHeap(int[] d, int heapsize) {
    for(int i=(heapsize-1)/2;i>=0;i--)
        heapify(a,i,heapsize);
}
```

A heapify kupacba rendező függvény első lépésben meghatározza a jobb és a baloldali részfához tartozó tömbindexeket (left() és right() függvények). A következő két feltétellel meghatározza, hogy az i, a jobb és a baloldali részfa közül melyik tartalmazza a legnagyobb értéket. Ha ez az érték nem az i csúcsban található, akkor ezzel megtörténik az értékek cseréje. Mivel a csere után előfordulhat, hogy a részfában elromlik a kupac tulajdonság, ezért erre a részfára elindítunk egy heapify függvényhívást, ami szükség esetén javítja a részfát.

```
void heapify(int[] d, int i, int heapsize){
    int lt=left(i);
    int rt=right(i);
    int great,tmp;
    if(lt<heapsize && d[lt]>d[i]) great=lt;
    else great=i;
    if(rt<heapsize && d[rt]>d[great]) great=rt;
```

```
        if(great!=i){
            tmp=d[i];
            d[i]=d[great];
            d[great]=tmp;
            heapify(d,great,heapsize);
        }
    }
```

A 19. ábrán a kupac építés során végzett lépéseket követhetjük nyomon.



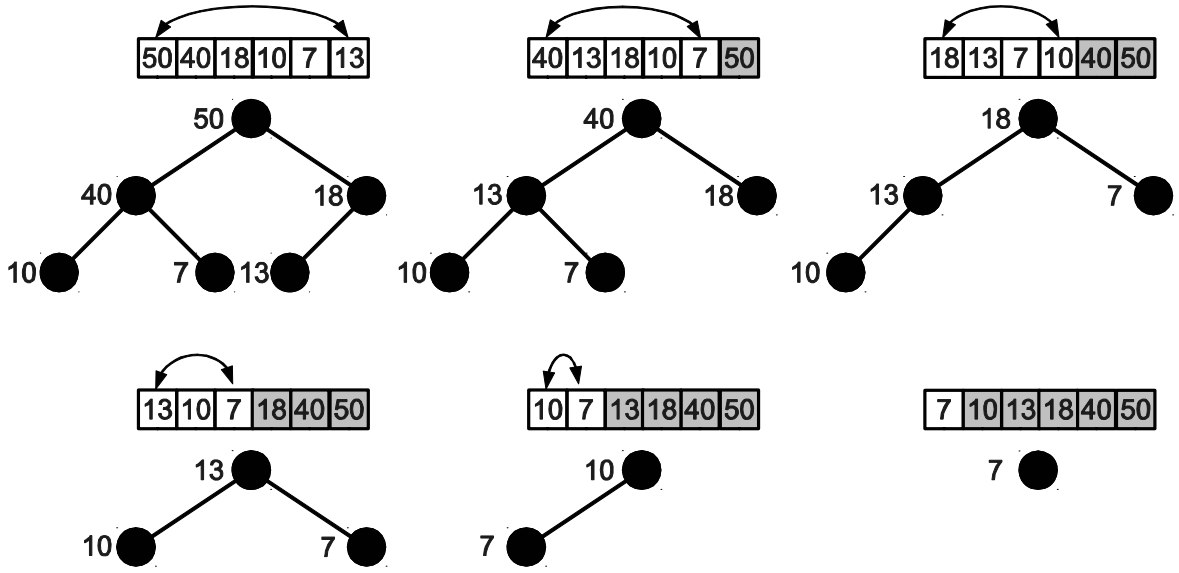
19. ábra: A kupac építés lépéseinek szemléltetése

A kupac építésére bemutatott függvényeket használhatjuk a kupac alapú rendező algoritmushoz is. A kupac rendezés a korábbi négyzetes komplexitású algoritmusokhoz képest hatékonyabb, az összehasonlításon alapuló algoritmusok közül a leghatékonyabb algoritmusok közé tartozik, komplexitása $O(n \log n)$. A kupacrendezés C++ nyelven a következőképpen implementálhatjuk.

```
void heapSort(int[] d, int heapsize){
    int tmp;
    buildHeap(d, heapsize);
    for(int i=heapsize;i>0;i--){
        tmp=d[0];
        d[0]=d[heapsize-1];
```

```
            d[heapsize-1]=tmp;
            heapsize=heapsize-1;
            heapify(d,0,heapsize);
        }
    }
```

A következő példában rendezzük az előzőekben kupaccá alakított 50, 40, 18, 10, 7, 13 elemeket tartalmazó tömböt a kupacrendezés algoritmussal. Az algoritmus lépéseit a 20. ábrán láthatjuk.



20. ábra: A kupacrendezés lépéseinek bemutatása egy példa segítségével

Gyorsrendezés

A gyorsrendezés (*quick sort*) az „oszd meg és uralkodj” elvén rendezi az elemeket. Ez azt jelenti, hogy a tömböt több kisebb részre osztja, és a kisebb részeket rendezi. A kisebb részek rendezésével előáll az egész rendezett tömb. Az oszd meg és uralkodj elvet több algoritmus is használja, az algoritmusok működésében a különbség általában a megosztás elvében való különbözőségből fakad.

A gyors rendezés során az eredeti tömböt két résztömbre osztjuk. Az egyik résztömb egy kulcs (vagy más néven pivot) elemnél nem nagyobb, a másik résztömb pedig a nagyobb elemeket tartalmazza. Ha elvégezzük a két résztömb rendezését, akkor a rendezett résztömbök egymáshoz fűzésével az eredeti tömböt kapjuk vissza rendezve. Természetesen a résztömbök rendezését is meg kell oldani valahogy, amit ismét két-két még kisebb résztömbre osztással végezhetjük el. Ez a kisebb tömbökre való particionálás egészen addig folytatható, amíg egy elemű résztömböket kapunk. A gyors rendezés hatékonysága a particionálás jóságának, azaz a pivot elem kiválasztásának a függvénye.

A tömb particionálása a pivot elem meghatározásából és a tömbelemek a pivot érték alapján való szétválogatásából áll. Egy jó pivot érték megtalálása nem triviális feladat. Akkor jó

egy pivot érték, ha a keletkező résztömbök közel azonos méretűek lesznek. Sok módszer és ajánlás van a pivot értékhez. A legegyszerűbb megoldás, ha a tömb első elemét választjuk pivot értéknek. Ennél jobb pivot érték, ha több tömbelem átlagát választjuk pivotnak. A tömb particionálása a pivot meghatározása után az elemek cseréjével folytatódik. Ehhez a tömbben jobbról és balról is elindítunk egy-egy indexet. A jobb oldali indexet (right) addig csökkentjük, míg olyan értéket találunk a tömbben, ami kisebb, mint a pivot, vagy elértük a baloldali indexet. A baloldali indexet (left) addig növeljük, míg egy nagyobb értéket találunk mint a pivot, vagy elértük a másik indexet. Az indexekhez tartozó tömbelemeket kicseréljük. A tömböt az indexek alapján osztjuk két résztömbre.

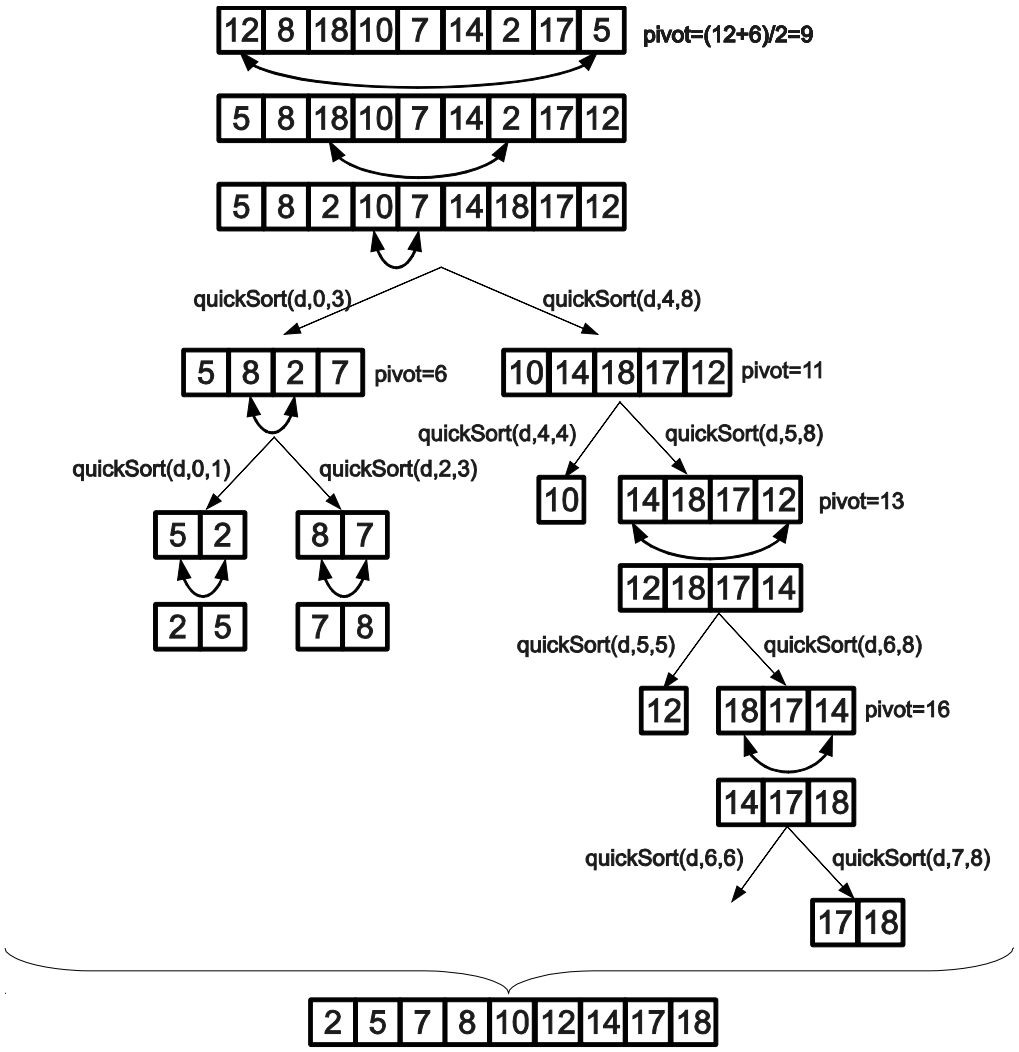
A következő gyorsrendező algoritmus paraméterei a tömb, amit rendezni kell és a tömb első és utolsó elemének az indexei. Az algoritmus az indexek közötti tömbelemeket rendezi. Pivot értéknek az aktuális tömb első és utolsó elemeinek a számtani közepét választjuk.

```
void quickSort(int d[], int first, int last){
    int left=first, right=last, tmp;
    double pivot=(d[first]+d[right])/2.0;

    do{
        while(d[left]<pivot)
            left++;
        while(d[right]>pivot)
            right--;
        if(left<right){
            tmp=d[left];
            d[left]=d[right];
            d[right]=tmp;
            left++;
            right--;
        }
    }while(left<=right);

    if(first<right)
        quickSort(d,first,right);
    if(left<last)
        quickSort(d,left,last);
}
```

A gyorsrendezés átlagos esetben $O(n \log n)$ komplexitású, legrosszabb esetben pedig négyzetes komplexitású algoritmusok közé tartozik. A legrosszabb eset akkor áll elő, ha a pivot elem kiválasztása után az egyik résztömb mérete az algoritmus futása során folyamatosan 1. A 21. ábrán a gyorsrendezés lépéseinek a bemutatását láthatjuk egy kilenc elemű tömb segítségével.



21. ábra: A gyorsrendezés lépéseinek bemutatása egy mintafeladat segítségével

Összefésüléses rendezés

Az összefésüléses rendezés (*merge sort*) is az „oszd meg és uralkodj” elven rendezi a tömböt. Az algoritmus alapötlete abból ered, hogy ha van két rendezett résztömbünk, akkor azokból viszonylag könnyű egy olyan rendezett tömböt kapni, mely tartalmazza mindegyik résztömb elemét. Ez a művelet az összefésülés művelete.

Legyen a két rendezett tömbünk A és B. Az A tömb n, a B tömb m elemet tartalmaz. Az összefésülés művelete során az A és a B tömb elemeit akarjuk tárolni a C tömbben, mely mérete n+m. Az első lépésben a C tömb első elemét szeretnénk meghatározni. Nyilvánvalóan a C tömb első elem vagy az A, vagy a B tömb legkisebb eleme lesz. Ezek az elemek az A és a B tömb legelső indexén találhatóak, melyek közül a kisebbet írjuk át a C tömb első helyére. Az átírt értéket a továbbiakban nem vesszük figyelembe. A következő lépésben a C tömb második elemének meghatározása lesz, amihez ismét az A és a B tömb aktuálisan legkisebb elemeit kell megvizsgálni. Minden egyes összehasonlítás után egy elem a helyére kerül, kivéve az utolsót, amivel két elem helyét rögzítjük.

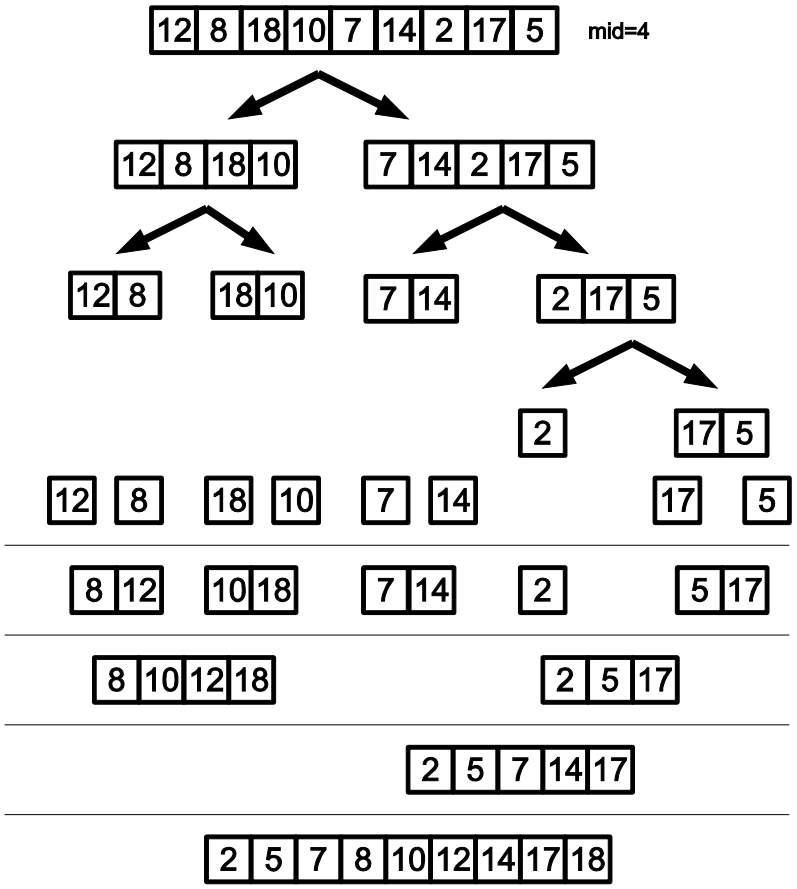
Az összefésülés hatékony művelet, hiszen a tömbök egyszeri végigolvasásával megkapjuk a rendezett sorozatot ($O(n)$). Az összefésülés hatékonysága miatt több más algoritmus használja ezt az ötletet.

Az összefésüléses rendezés is használja az összefésülés műveletet. A rendezés rekurzión alapul, mely során rendezzük összefésüléses rendezéssel a tömb első felét és a második felét külön-külön, majd utána fésüljük össze a rendezett résztömböket. Az algoritmus pszeudo kódja a következő:

```
void MergeSort(d, i, j) {
    if (i < j) {
        mid = [(i + j) / 2];
        MergeSort(d, i, mid);
        MergeSort(d, mid+1, j);
        Osszefesul(d, i, mid, j);
    }
}
```

Az összefésülő rendezés egészen addig bontja kisebb részekre a tömböt, míg a vizsgált tömb egy elemű lesz (az egy elemű tömb rendezettnek tekinthető). A résztömbökre bontáshoz a mid változóval meghatározza a tömb középső elemét. Ha a eljutott az algoritmus a rekurzióban egy elemű tömbökhöz, akkor elkezdi összefésülni azokat az Osszefesul() függvény segítségével. A függvény paraméterei a d tömb, és az indexek, amik a rendezett résztömböket jelölik. Az összefésüléshez egy segéd tömb használata szükséges, ami miatt az összefésüléses rendezés nem helyben rendező algoritmus.

Az összefésülő rendezés komplexitása legrosszabb esetben $O(n \log n)$. A gyorsrendezésnél bemutatott példát rendezzük az összefésülő rendezés segítségével. A példa és az összefésülő rendezés lépései a 22. ábrán látható. Az ábrán követhetjük a tömb kisebb részekre bontásának lépéseit, majd utána a rendezett részek összefésülését.



22. ábra: Összefésülő rendezés lépéseinek bemutatása egy mintafeladat segítségével

Láda rendezés

A láda rendezés (*bin packing, bin sorting*) egy olyan rendező algoritmus, amit csak néhány különleges esetben lehet használni. A rendezés úgy működik, mintha lenne M darab ládánk és n darab tárgyunk, amit valami érték alapján rendezni kell. Tudjuk, hogy a tárgyak legfeljebb M különböző értéket vehetnek fel. Ha a ládáinkat felcímkézzük a lehetséges értékek alapján, és a ládákat rendezzük az érték szerint, akkor ha kapunk egy rendezendő tárgyat, akkor rögtön abba a ládába helyezhetjük el, ahova a felcímkézés alapján tartozik.

A rendezés elve szerint tehát az A tömb rendezéséhez végig kell haladni az A tömb elemein, és az A tömb értékével megcímezhetjük azt a ládát (Lada), melyben az értéket el kell helyeznünk. Ha az A tömb több ugyanolyan értéket tartalmazhat, akkor egy ládába többször is bele kerülhetnek ugyanolyan értékű tömbelemek. Ekkor az ütközés feloldására minden ládát egy láncolt lista segítségével implementáljunk.

Az algoritmust a BinSort függvényben implementáltuk. A függvény paraméterként a rendezendő tömböt, a ládák tömbjét és a rendezendő elemek számát kapja. Az algoritmus két ciklusból épül fel. Az első ciklus feladata a ládák ürítése, a második ciklus pedig a rendezendő elemek ládákba pakolását tartalmazza. Az algoritmus futási ideje $O(n + M)$. Általában az $n>M$ feltétel teljesül, így a rendezés $O(n)$ jellegű algoritmus. Láthatjuk, hogy ez egy hatékony rendezési algoritmus abban az esetben, ha a bemenetre teljesülnek a kívánt feltételek.

```
void BinSort( int *A, IntList *Lada, int n ) {
```

```
    int i;
    for(i=0;i<M;i++)
        Lada[i] = EMPTY;
    for(i=0;i<n;i++)
        Lada[a[i]].addToHead(a[i]);
    //ládák tartalmának összepakolása egy tömbbe
}
```

Gráfok

A gráf egy olyan adatszerkezet, mely csúcsokat/csomópontokat (*node*) és a csúcsok között értelmezett éleket/összeköttetéseket (*arc*) tartalmaz. A gráfok sokoldalú eszközök, mellyel többfajta feladatot lehet ábrázolni és segítségével megoldani. A gráfelmélet a matematika és a számítástechnika fontos területe. A gráfokat kategorizálhatjuk az élek és a csúcsok tulajdonságai alapján.

A G gráfot a csúcsok és az élek halmaza definiálja. A $G=(V, E)$ egy egyszerű gráf, ha a V nemüres halmaz a csúcsokat tartalmazza, az E halmaz pedig az éleket. Alapesetben egy gráf irányítatlan, azaz nem különböztetjük meg, hogy egy benne szereplő (i, j) él az i csúcsból a j csúcsba, vagy a j csúcsból az i csúcsba vezet. Irányított gráf esetén megkülönböztetjük az élek irányát. A multigráf esetén a gráfban találhatunk hurokéleket, vagy többszörös élet. A hurokél olyan él, melynek kezdő- és végcsúcsa ugyanaz.

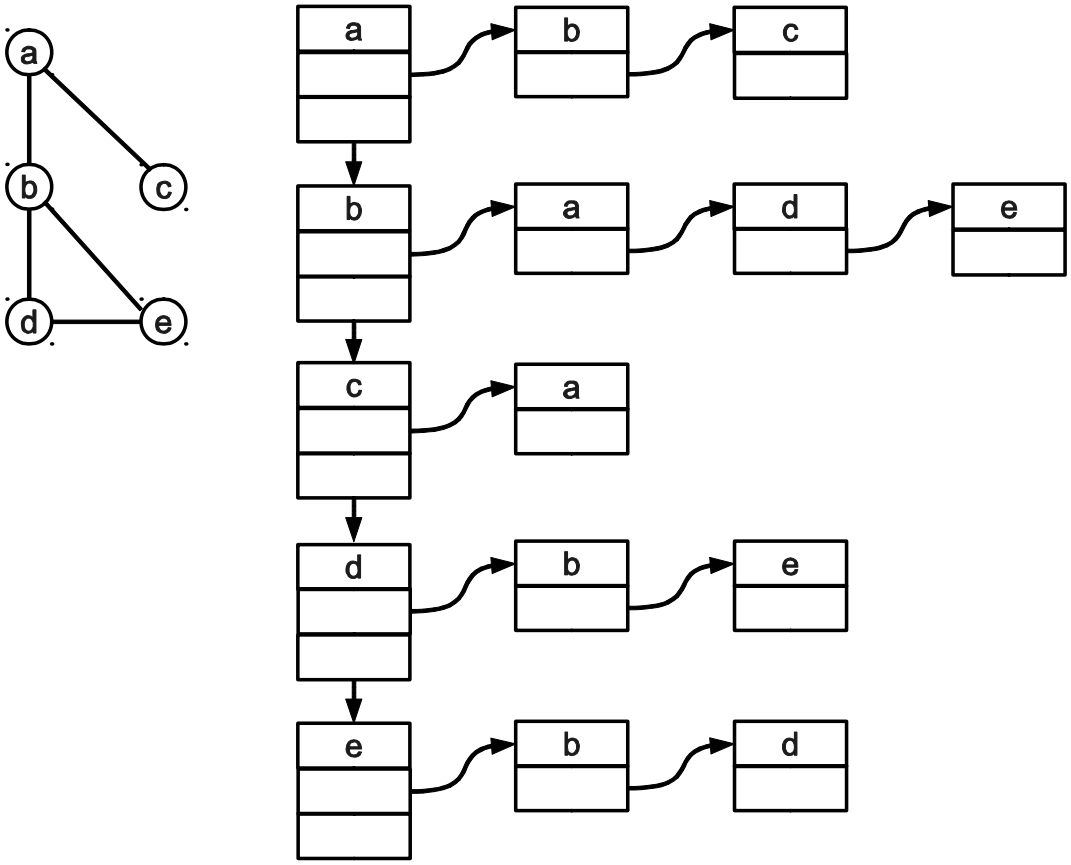
A gráfban két csúcs szomszédos, ha van köztük él. A gráfban útnak nevezzük szomszédos csúcsok sorozatát. Az olyan utat, melynek kezdő és vég csúcsa ugyanaz körnek nevezzük. Egy gráf összefüggő, ha bármely két csúcs között van út. A gráf súlyozott, ha minden éléhez hozzárendelünk egy értéket. Ez az érték alkalmazástól függően az él hossza, súlya, vagy költsége is lehet. A gráfban az élek és a csúcsok azonosítására címkéket rendelhetünk hozzájuk.

Gráfok ábrázolása

Gráfok tárolására többfajta módszer és adatszerkezet is használható. Ha ábrázolni akarjuk a gráfot tárolni kell a gráf csúcsokat és az éleket és a közöttük levő kapcsolatot. A tárolásra használható lehetőségek amiket bemutatunk az a szomszédsági (adjacencia) lista és mátrix, valamint az incidencia mátrix.

Adjacencia lista és mátrix

Az adjacencia lista során egy-egy láncolt listában tároljuk a gráf csúcsait és a csúcsok szomszédait, vagyis azokat a csúcsokat amik között az élek vezetnek. Az adjacencia lista két szintű láncolt listából áll. Az egyik láncolt lista tartalmazza a gráf csúcsait és minden csúcselemből egy-egy láncolt lista épül fel, mely listák a szomszédos csúcsokat tartalmazzák. Irányítatlan gráf esetén ha egy i csúcs szerepel a j csúcs szomszédsági listájában, akkor a j csúcs is szerepel az i szomszédsági listájában. Ha a gráf irányított, akkor az irányításnak megfelelően csak az egyik csúcs szomszédsági listája tartalmazza a másik csúcsot. A 23. ábrán egy példát láthatunk egy egyszerű irányítatlan gráf adjacencia listájára.



23. ábra: Gráf ábrázolása adjacencia listával

Az adjacencia listát mátrix formájában is fel lehet írni. Az adjacencia mátrix egy olyan $|V|$x$|V|$ méretű bináris mátrix, melynek a_{ij} elemét a következő kifejezés alapján számolhatjuk ki:

$$a_{i,j} = \begin{cases} 1, & \textit{ha } i \textit{ és } j \textit{ csúcsok között van él} \\ 0, & \textit{különben} \end{cases}$$

A kifejezés alapján egyszerű (irányítatlan) gráfokra, a gráf adjacencia mátrixa szimmetrikus. Ha a gráf hurokélt tartalmaz, akkor azt a mátrix főátlójában tárolhatjuk. A 23. ábrán látható gráf adjacencia mátrixa a következő:

	a	b	c	d	e
a	0	1	1	0	0
b	1	0	0	1	1
c	1	0	0	0	0
d	0	1	0	0	1
e	0	1	0	1	0

Az adjacencia mátrix általánosításával lehetőség van többszörös éleket, vagy súlyozott éleket tartalmazó gráfok felírására.

Incidencia mátrix

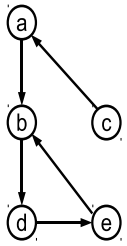
Az incidencia mátrix a gráf csúcsai és élei közötti kapcsolatokat tartalmazza. Ahhoz, hogy az incidencia mátrixot felírjuk azonosítani/címkézni kell a gráf csúcsait és éleit. Az incidencia mátrix egy olyan $|V|$x$|E|$ méretű bináris mátrix, melynek a_{ij} elemét a következő kifejezés alapján számolhatjuk ki:

$$a_{i,j} = \begin{cases} 1, & \textit{ha i csúcs illeszkedik a j élre} \\ 0, & \textit{különben} \end{cases}$$

A 23. ábrán látható gráf incidencia mátrixa a következő:

	(a,b)	(a,c)	(b,d)	(b,e)	(d,e)
a	1	1	0	0	0
b	1	0	1	1	0
c	0	1	0	0	0
d	0	0	1	0	1
e	0	0	0	1	1

Egyszerű gráfokra, melyben minden él két csúcsra illeszkedik az incidencia mátrixban az oszlopok összege kettő.



24. ábra: Irányított gráf

A 23. ábrán látható irányított gráf incidencia mátrixában 0-ák és 1-ek szerepelnek, mert nem kell megkülönböztetnünk a csúcsokat és az illeszkedő éleket az él iránya alapján. Egy irányított gráf incidencia mátrixában azonban ábrázolni kell, hogy az adott csúcsnál az él befele, vagy kifele mutat. Ezt az irányt az 1 és -1 értékekkel jelölhetjük. A 24. ábrán látható irányított gráf incidencia mátrixa a következő:

	(a,b)	(a,c)	(b,d)	(b,e)	(d,e)
a	-1	1	0	0	0
b	1	0	-1	1	0
c	0	-1	0	0	0
d	0	0	1	0	-1
e	0	0	0	-1	1

Irányított gráfokra az incidencia mátrixban az oszlopok összege kettő.

Gráf bejárása

Hasonlóan, mint egy fa bejárás, gráfok bejárása is fontos algoritmikus feladat. A gráf bejárásán a csúcsok éleken keresztüli pontosan egyszer való meglátogatását értjük. A fáknál bevezetett szélességi és mélységi bejárások egy az egyben nem használhatóak gráfokra, mivel egy gráf tartalmazhat kört és hurokélt is (ellentétben a fával, ami definíció szerint körmentes), ami a fa bejárásra kidolgozott algoritmusok esetén végtelen ciklust eredményezne. Hogy elkerüljük a végtelen ciklust, meg kell jegyezni, hogy mely csúcsokat látogattuk már meg, hogy az újbóli látogatást elkerüljük.

Gráf mélységi bejárása

Mélységi bejárás (*DFS*, *depth-first search*) egy módszer a gráf csúcsainak bejárására. A mélységi bejárás algoritmus alapötletét sok más informatikai algoritmusban megtalálhatjuk. Az alapötlet röviden: haladjunk tovább (lefele) egészen addig, míg lehetséges; ha már nem lehet továbbhaladni, akkor lépjünk vissza egészen addig míg valamely új irányba el nem tudunk indulni.

A mélységi bejárás közben a csúcsok kétfajta állapotban lehetnek: FEHÉR: azok a csúcsok, melyekben még nem jártunk, FEKETE: azok a csúcsok melyeket már meglátogattunk. Kezdetben az összes csúcs színe FEHÉR. Az algoritmus tetszőleges csúcsból indulhat. Egy rekurzív lépés során a következők történnek:

1. Fessük be FEKETÉRE azt az *i* csúcsot, ahol az algoritmus tart
2. Minden olyan (*i*, *j*) élre, ahol a *j* csúcs színe FEHÉR indítsunk el egy mélységi keresést

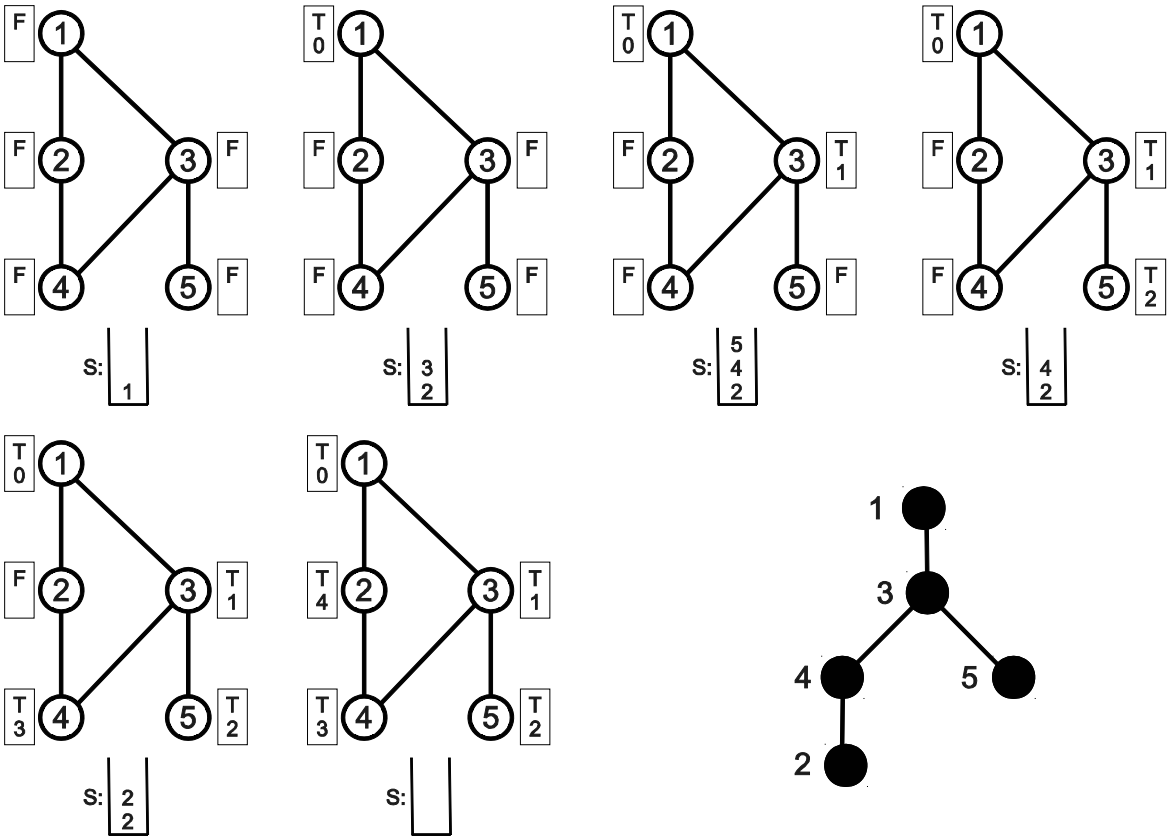
A rekurzív függvényszervezés helyett verem alkalmazásával létrehozhatunk egy olyan iteratív algoritmust, mely ugyanilyen mélységi jellegű gráfbejárást eredményez. Tegyük fel, hogy N darab csúcs található a gráfban. A program az ADJ-vel jelölt *N*x*N* méretű egészeket tartalmazó tömbbel adjacencia mátrixként tárolja a gráf adatait. Az algoritmust egészítsük ki egy számlálóval (step), amivel felcímkézhetjük a csúcsokat a bejárás sorrendjében. Ezeket a címkéket a d tömbben tároljuk. Az algoritmus indulásakor fessük az összes csúcsot FEHÉRRE oly módon, hogy a visited tömb értékeit hamisra állítjuk. Az algoritmus megvalósítható úgy is, hogy bevezetünk egy harmadik állapotot (pl. SZÜRKE) azokra a csúcsokra, melyek aktuálisan a veremben vannak, azért hogy ne kerülhessenek be többször. A jelenlegi megvalósításban többször is bele kerülhetnek, azonban a veremből való kikerülés alkalmával ellenőrizzük, hogy feldolgoztuk-e már a kikerülő csúcsot.

```
IntStack S;
int i,j,step,start_node;
step=0;
S.push(start_node);
while(!S.isEmpty()) {
    i=S.pop();
    if(visited[i]==false){
        visited[i]=true;
        d[i]=step++;
        for(j=0;j<N;j++){
            if(ADJ[i][j]==1 && visited[j]==false)
            S.push(j);
```

```
            }
        }
    }
```

Kezdetben a verembe tegyük bele azt a csúcsot, ahonnét a bejárást indítjuk (start_node). Az iteratív algoritmus addig tart, amíg a verem értéket tartalmaz. Ha kiveszünk a veremből egy csúcsot és csúcsot még nem látogattuk meg, akkor a csúcsot látogatottnak jelöljük (visited[i]=true) és a d tömbben tároljuk, hogy hányadikként jártuk be ezt a csúcsot (d[i]=step++). Végezetül azokat a szomszédos csúcsokat, amiket még nem jártunk be belerakjuk a verembe (S.push(j)).

Az algoritmus végeztével a visited tömbben jelölve vannak azok a csúcsok amik éleken haladva elérhetőek a start_node csúcsból. A d tömb tartalmazza az elérhető csúcsok sorrendjét. Az algoritmus lépéseit a 25. ábrán láthatjuk. Az ábra mutatja a verem és lépésenként a visited és a d tömb tartalmát. A keresés az 1-es számú csúcsból indult.



25. ábra: A mélységi kereső algoritmus lépéseinek a bemutatása

Ha a gráf irányított és körmentes, akkor a mélységi bejárás használható a gráf topologikus rendezésére. A gráf topologikus rendezésekor a gráf csúcsait kell sorba rakni, oly módon, hogy ha két csúcs között van irányított él, akkor a kezdő csúcs nem előzheti meg az élhez tartozó végcsúcsot. Topologikus rendezésnél gyakorlatilag a gráf csúcsait úgy kell egy vonal mentén felsorakoztatni, hogy a vonal irányát tekintve a csúcsok között az élek csak egy irányba mutathatnak.

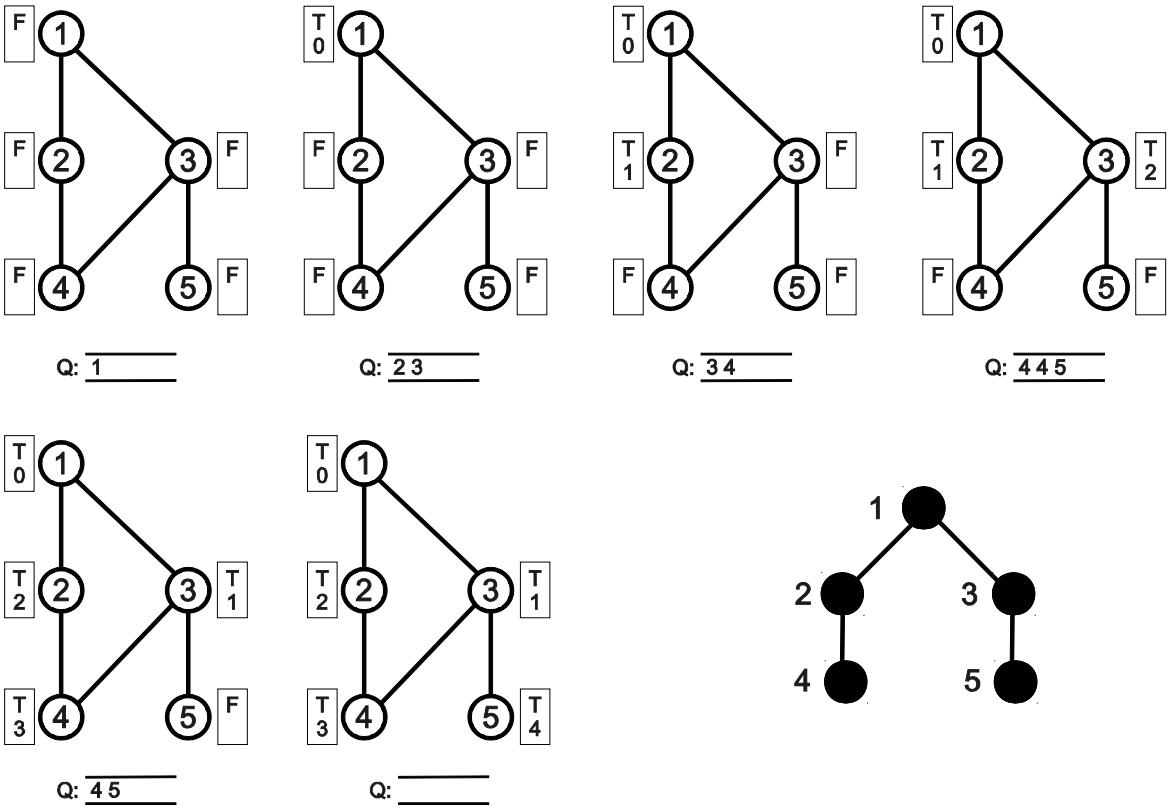
Gráf szélességi bejárása

A mélységi bejáráshoz hasonlóan a fákra bemutatott szélességi bejárás is kiterjeszthető gráfokra is. Míg a mélységinél egy verem segítségével tároltuk a következő lépésekben bejárandó csúcsokat, addig a szélességi bejárásnál egy sor segítségével tudjuk a bejárást vezérelni. A mélységi bejárásnál bevezetett változókkal a szélességi keresés a következőképpen írható le.

```
IntQueue Q;
int i,j,step,start_node;
step=0;
Q.enqueu(start_node);
while(!Q.isEmpty()) {
    i=S.dequeue();
    if(visited[i]==false){
        visited[i]=true;
        d[i]=step++;
        for(j=0;j<N;j++){
            if(ADJ[i][j]==1 && visited[j]==false)
                Q.enqueue(j);
        }
    }
}
```

A szélességi bejáró algoritmus egy sorban tárolja azokat az egész értékeket, melyek a gráf csúcsait azonosítják (IntQueue Q). A step változót használjuk arra, hogy eltárolhassuk, melyik lépésben/iterációban látogattunk meg egy adott gráf csúcsot. Ez a változó iterációnként eggyel növekszik. A start_node változót használjuk arra, hogy a bejárást valamely gráf csúcsból elindíthassuk. Kezdetben a start_node változót rakjuk bele a Q sorba. A bejárás addig tart, míg van csúcs a sorban, azaz van olyan potenciális csúcs, melyből még valamely irányban tovább tudunk haladni.

A ciklus magjában végezünk el egy bejárási lépést. Ebben a lépésben eltávolítjuk a sor legelső elemét (i csúcs). Ha ebben a csúcsban még nem jártunk, akkor mostantól látogatottnak minősítjük (visited[i]=true) és eltároljuk hogy a csúcsba hányadik lépéssel jutottunk el (d[i]=step++). A következő ciklussal végignézzük azokat a csúcsokat, melyeket még nem látogattunk meg és szomszédosak az i csúccsal, majd belerakjuk ezeket a sor végére (Q.enqueue(j)).



26. ábra: Szélességi kereső algoritmus lépéseinek a bemutatása

A 26. ábrán egy példát láthatunk a szélességi kereső algoritmus működésére. Az ábrán láthatjuk, hogy lépésenként hogyan változik a Q sor tartalma, melyek azok a csúcsok, melyekben jártunk már (T/F érték), és hányadik lépésben értünk el egy adott csúcshoz. Az ábrán láthatjuk a bejáráshoz tartozó fát, mely azokat a gráféleket tartalmazza, melyeket használtunk a csúcsok felderítése közben. A keresés az 1-es számú csúcsból indult. Összehasonlítva a mélységi bejárásnál kapott fával szemléletesen láthatjuk az algoritmusok jellegét. Míg a mélységi bejárásnál hosszabb utak alakulnak ki és a fa több szintet tartalmaz (mélyebb), addig a szélességi bejárásnál a fa vízszintesen növekszik és rövidebb utakat tartalmaz csak.

Körkeresés

Vannak olyan feladatok, melyekhez tartozó gráfban azt kell eldöntenünk, hogy a gráf körmentes-e, vagy sem. A mélységi bejárás algoritmusát használhatjuk ennek a problémának a megoldására. Általában ez a jellegű feladat irányított gráfok esetén jelentkezik. Azokat a gráfokat, melyekben nincsen irányított kör körmentes irányított gráfoknak (*KIG, DAG, directed acyclic graph*) nevezzük.

A gyakorlatban körkeresési feladatokat operációs rendszerekben, adatbázis kezelő rendszerekben, gyártó folyamatok tervezése közben találhatunk. Operációs rendszerek esetén, ha folyamatok egymásra várakozását egy-egy éllel ábrázoljuk egy irányított gráfban, akkor abban az esetben, ha az A folyamat B-re, a B folyamat C-re, a C folyamat pedig az A-ra várakozik egy kört kapunk az irányított gráfban. Az operációs rendszernek ezt az állapotát holtpontnak nevezzük.

A mélységi gráf bejárás algoritmust használhatjuk arra, hogy eldönthessük hogy a gráf tartalmaz-e kört, vagy nem. Az algoritmusban be kell vezetni egy lépésszámlálót, majd minden

csúcshoz meg kell jegyezni, hogy melyik az a lépés, amikor először eljutottunk a csúcsba, és melyik az a lépés, amikor a csúcsban utoljára jártunk (befejezési szám), azaz a csúcsból épült részfát befejeztük. Ezekkel a lépésszámokkal osztályozhatjuk a gráf (x, y) éleit a következő kategóriákba:

- fa élek: azok az élek, melyek az algoritmus futása során a mélységi fa részei lettek
- visszaélek: azok az élek, melyek a fa alacsonyabb szintjén szereplő csúcsából halad a fa magasabb szintjén levő csúcsába, tehát a fában az x csúcs az y leszármazottja
- előreél: azok az élek, melyek a fa magasabb szintjén szereplő csúcsából halad a fa alacsonyabb szintjén található csúcsába, tehát a fában az y csúcs az x leszármazottja
- keresztél: azok az élek, melyek olyan csúcsokat kötnek össze, melyek egyike sem őse a másiknak.

Ha meghatároztuk minden élre a típusát, akkor meg lehet határozni, hogy a gráf körmentes-e. A gráfban nem találunk visszaélt pontosan akkor, ha a gráf körmentes.

Az algoritmus során kapott befejezési számok a fa topologikus rendezésére is használhatjuk. A topologikus rendezés egy gráf esetén az csúcsok egy olyan sorrendje, melyre ha (x,y) él része a fának, akkor az x csúcs sorrendben előrébb van, mint az y csúcs. A mélységi kereséssel kapott befejezési számok alapján csökkenő sorrendben írjuk ki a csúcsokat, akkor egy topologikus rendezést kapunk.

Erősen összefüggő komponensek meghatározása

Egy irányított gráf erősen összefüggő, ha a benne szereplő bármely két csúcspár között létezik irányított út. Általában az egész gráfra nem mondható el, hogy erősen összefüggő, akkor is érdekes lehet az erősen összefüggő komponensek (részgráfok) meghatározása. Ha megvannak az erősen összefüggő komponensek, akkor az eredeti gráf egy redukcióját hajthatjuk végre létrehozva az úgynevezett komponens gráfot. Az erősen összefüggő komponensek lesznek a komponens gráf csúcsai, és a komponensek között akkor lesz él, ha az egyik komponens valamely i csúcsából vezet él a másik komponens valamely j csúcsába. A komponens gráf vizsgálatával eldönthető például, hogy egy irányított gráf félig összefüggő-e.

Az erősen összefüggő komponensek meghatározására a mélységi bejárás algoritmust használhatjuk. Az algoritmus a következő lépésekből épül fel:

- Mélységi bejárást hajtunk végre a gráfon (befejezési sorszámok a csúcsokhoz)
- Megfordítjuk az irányított élek irányítását
- Mélységi bejárással végigmegyünk a megfordított gráfon a legnagyobb befejezési sorszámmal rendelkező csúcsból indulva. Azok a csúcsok, amiket elérhetünk ebből a csúcsból azok a részei egy erősen összefüggő komponensnek. A következő mélységi bejárást indítsuk a maradék csúcsok között szereplő legnagyobb sorszámúból.

Legrövidebb út keresés

A legrövidebb út keresés (*SPA*, *shortest path algorithm*) klasszikus gráf algoritmusok közé tartozik, amire sokfajta megoldást találhatunk. A legrövidebb út keresést általában súlyozott gráfokon értelmezzük. Az ilyen gráfokban egy él súlya általában a csúcspontok (pl. városok) távolságát jelöli.

A legrövidebb utat általában két csúcs között kell meghatározni azonban a legrövidebb út kereső algoritmusok egy kezdőcsúcsból indulva az összes elérhető csúcsba megkeresi az oda vezető legrövidebb utat. A „legáltalánosabb” legrövidebb út kereső algoritmusok bármely két csúcspár között megadják a legrövidebb utat és annak hosszát (pl. a Floyd-Warshall, vagy a Johnson algoritmus).

A legrövidebb út kereső algoritmusokat csoportosíthatjuk az alapján, hogy milyen típusú éleket tartalmaz a súlyozott gráf, amit vizsgálunk. Vannak olyan algoritmusok (pl. Dijkstra), mely csak nemnegatív éleket tartalmazó gráfokon használható. Ha a gráf negatív élet is tartalmaz, akkor például a Bellman-Ford algoritmus használható. Ha a gráf tartalmaz negatív élt, akkor a legrövidebb út létezésének szükséges feltétele, hogy a gráfban ne legyen olyan kör, mely körben levő élek összege negatív. Ha ilyen kör szerepelne egy gráfban, akkor a kör mentén haladva folyamatosan csökkenthetnénk a legrövidebb út hosszát, ami nyilván azt jelenti, hogy a feladatnak ebben az esetben nincsen megoldása. Ebben az alfejezetben a Dijkstra és a Bellman-Ford algoritmus kerül bemutatásra.

Dijkstra-algoritmus

A továbbiakban olyan legrövidebb út kereső algoritmusokkal foglalkozunk, ahol az egy kezdőcsúcsból induló legrövidebb utak meghatározása a cél csak nemnegatív éleket tartalmazó súlyozott gráfokra. A legrövidebb út kereső algoritmusokban a két csúcs közötti legrövidebb út meghatározásához a köztes csúcsokba számolt legrövidebb út távolságát és a legrövidebb úton szereplő éleket/csúcsokat is tárolni kell. Ezt az információt eltárolhatjuk egy csúcshoz rendelt távolság címke segítségével és annak a csúcsnak a megjegyzésével, amiből a hozzá vezető legrövidebb út megy. A legrövidebb út kereső algoritmus ezekkel a távolság címkékkel dolgozik futás közben.

Dijkstra algoritmusa egy kezdőcsúcsból az összes elérhető csúcsba meghatározza a legrövidebb utat. Az algoritmus során kétfajta csúcstípust különböztetünk meg: vannak olyan csúcsok, melyekbe már ismerjük a legrövidebb utat (rögzített csúcsok) és azok a csúcsok, amibe még nem tudjuk az út hosszát, vagy mert nem jártunk még ebben a csúcsban, vagy pedig még úgy gondoljuk hogy a későbbiek során találhatunk ebbe a csúcsba a jelenleginél rövidebb utat is. Az algoritmus azokba a csúcsokba, amelyek még nem rögzítettek folyamatosan javítások végez, oly módon hogy újabb legrövidebb utakat találhat hozzá.

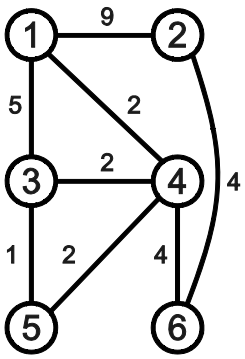
A Dijkstra algoritmusának implementálásához vegyünk egy N csúcsot tartalmazó gráfot. A gráfban szereplő élek súlyát tároljuk az NxN méretű ADJ adjacencia mátrixban. Ha két csúcs között nincs él a gráfban, akkor az adjacencia mátrix megfelelő értékét állítsuk végtelenre.

Az algoritmus a futás során rögzíti a gráfban szereplő csúcsokat. Minden lépésben egy csúcs rögzítésére kerül sor annak ábrázolására, hogy mely csúcsok kerültek már rögzítésre használjuk a fix tömböt, mely kezdetben minden csúcsra false értéket tartalmaz. Hasonlóan a fix tömbhöz vezessünk be egy másik tömböt a távolság tárolására (distance – kezdetben végtelen) és a csúcsba vezető legrövidebb út melyik előző csúcson keresztül megy (pred) tárolására.

A Dijkstra-algoritmus a start_node csúcsból indítja a keresést. Minden egyes lépésben egy csúcs rögzítésére kerül sor (for ciklus egy iterációjában). Az algoritmus annak a csúcsnak a rögzítését végzi el, amibe már nem találhatunk rövidebb utat. Ehhez először meg kell keresni azt a csúcsot, melybe vezető út jelenleg a legrövidebb, azaz ki kell választani azt még nem rögzített csúcsot, melyhez tartozó distance értéke a legkisebb. Ezt a műveletet a GetClosestNode() függvény végzi el. Ezek után rögzítésre kerül a kiválasztott csúcs (fix[closest] = true), majd megnézzük hogy melyik csúcsokba lehet a rögzített (closest) csúcson keresztül javítani az eddigi legrövidebb út hosszát. Ha van olyan csúcs, amibe az aktuális legrövidebb út hosszabb, mint a closest-en keresztül vezető legrövidebb út (distance[closest] + ADJ[closest][j]), akkor javítjuk ennek a csúcsnak az értékét és a hozzá vezető legrövidebb út irányát (distance és pred változók). A Dijkstra algoritmus futási ideje $O(V^2+E)$, mivel tartalmaz két egybeágyazott ciklust a csúcsokra vonatkozóan, illetve a

legközelebbi csúcs meghatározása az élek számával arányos műveletet tartalmaz. Átlagos esetben az élek száma elhanyagolható a négyzetes taghoz képest.

```
distance[start_node]=0;
for(i = 0;i < N; i++) {
    closest=GetClosestNode();
    if(closest == -1) break;
    fix[closest] =true;
    for(int j=0;j<N;j++){
        if(distance[j]>distance[closest]+ADJ[closest][j]){
            distance[j]=distance[closest]+ADJ[closest][j];
            pred[j]=closest;
        }
    }
}
```



27. ábra: Legrövidebb út kereső mintafeladat a Dijkstra-algoritmushoz

A 27. ábrán látható gráf legrövidebb útjait szeretnénk meghatározni. A kezdőcsúcs, amiből a keresést indítjuk legyen a 3-as számú csúcs. Az algoritmus hat iterációban találja meg a csúcsokba vezető a legrövidebb utakat. A következő táblázatokban a pred, distance és fix vektorok alakulását láthatjuk.

i=0, closest=3	1	2	3	4	5	6
distance	5	∞	0	2	1	∞
fix	F	F	T	F	F	F
pred	3	-	-	3	3	-

i=1, closest=5	1	2	3	4	5	6
distance	5	∞	0	2	1	∞
fix	F	F	T	F	T	F
pred	3	-	-	3	3	-

i=2, closest=4	1	2	3	4	5	6
distance	4	∞	0	2	1	6
fix	F	F	T	T	T	F
pred	4	-	-	3	3	4

i=3, closest=1	1	2	3	4	5	6
distance	4	13	0	2	1	6
fix	T	F	T	T	T	F
pred	4	1	-	3	3	4

i=4, closest=1	1	2	3	4	5	6
distance	4	10	0	2	1	6
fix	T	F	T	T	T	T
pred	4	6	-	3	3	4

i=5, closest=2	1	2	3	4	5	6
distance	4	10	0	2	1	6
fix	T	T	T	T	T	T
pred	4	6	-	3	3	4

Bellman–Ford-algoritmus

Tegyük fel, hogy adott egy negatív negatív összköltségű irányított kört nem tartalmazó gráf, továbbá egy start_node kezdőcsúcs. Feladat az, hogy határozzuk meg, minden csúcsra, a hozzá vezető legrövidebb utat és annak hosszát!

A Bellman–Ford-algoritmus a Dijkstra-algoritmusnál megismert fokozatos közelítés műveletét végzi, azaz egy csúcson át a szomszédba vezető él mentén vizsgálja, hogy az illető él része-e a legrövidebb útnak, javító él-e. Egy lépésben az összes élre megvizsgálja, hogy javító él-e vagy sem, a csúcsokat nem rögzíti, hiszen a negatív súlyú élek miatt előfordulhat bármelyik csúcsnál javítás.

A Bellman–Ford-algoritmus pszeudó kódja az előző fejezetben bemutatott jelölésekkel a következőképpen néz ki:

```
distance[start_node]=0;
for(i = 0;i < N-1; i++) {
    minden (u,v) G-beli élre {
        if(distance[v]>distance[u]+ADJ[u][v]) {
            distance[v]=distance[u]+ADJ[u][v];
            pred[v]=u;
        }
    }
}
```

A Bellman–Ford-algoritmus csúcsszám*élszám darab iterációt hajt végre. Igazolható, hogyha a gráf nem tartalmaz negatív kört, akkor ennyi iteráció után nem lehet tovább javítani a legrövidebb utakat. Az algoritmus a Dijkstra-algoritmusnál is használt távolság tömböt használja a fokozatos javításra. Kezdetben ennek a tömbnek az elemeit végtelenre kell állítani. Az algoritmus végeztével a pred tömb segítségével visszakapjuk az összes csúcsba vezető legrövidebb utakat. Az algoritmus n-1 iterációban vizsgálja végig az éleket, így az algoritmus komplexitása ($O(N\,E)$)

Ha a ellenőrizni akarjuk, hogy a gráf tartalmazott-e negatív kört az algoritmus leállta után, akkor az összes élen végighaladva ellenőriznünk kell, hogy az élhez tartozó két csúcson lehet-e javítást végezni. Ha találunk olyan csúcsokat, ami között további javítást tehetünk, akkor a gráf negatív súlyú kört tartalmazott. Figyeljünk oda arra, hogy egyszerű (nem irányított), negatív súlyú élt tartalmazó gráf esetén az algoritmus a negatív súlyú élen oda-vissza lépkedve akármeddig javításokat tudna végrehajtani, így ilyen gráfokra az algoritmus nem használható.

Feszítő fa keresés

Egy gráf feszítő fájának (*spanning tree*) megtalálása fontos gráfelméleti feladat. A feszítő fa a gráfnak egy olyan részgráfja, ami a gráf minden csúcsát tartalmazza és fa. Általában a minimális feszítő fa (*MST*, *minimal spanning tree*) meghatározása a cél, ami a feszítő fák közül a legkisebb összesített élsúlyú.

Feszítő fa keresése sokfajta alkalmazást találhatunk. Ilyen feladatot kell megoldani ha például adott N darab város, bármely kettő között repülőjáratok üzemelnek. Az üzemeltető gazdasági okokból bizonyos városok között be szeretne zárni járatokat, azonban minden városnak elérhetőnek kell maradnia repülővel, ha máshogy nem átszállással. Ha a feladatot egy gráf segítségével ábrázoljuk, melyben a csúcsok a városoknak, az élek a távolságokat jelölik, akkor a minimális feszítő fa azokat a járatokat szolgáltatja, melyek üzemeltetésével az elérhetőség

megmarad és a repült távolságok összege minimális (utasforgalmi adatokat figyelmen kívül hagyva).

A korábbi fejezetekben bemutatott mélységi és szélességi kereső algoritmusok is egy feszítő fát keresnek a gráfhoz. Azonban ezek az algoritmusok nem biztos, hogy a minimális feszítő fát szolgáltatják. A minimális feszítő fa keresésre többfajta algoritmus létezik. Ezek közül a jellegüket tekintve a fontosabbak:

- Több fát bővítünk egészen addig míg egy összefüggő feszítőfát kapunk (pl. Kruskal algoritmus)
- Egy fát kezdünk el élekkel bővíteni egészen addig míg egy feszítőfát kapunk (pl. Jarnik-Prim algoritmus)
- Egy fát kezdünk el élekkel bővíteni, úgy hogy közben éleket is távolíthatunk el belőle egészen addig míg egy feszítőfát kapunk (pl. Jarnik-Prim algoritmus)

Kruskal-algoritmus

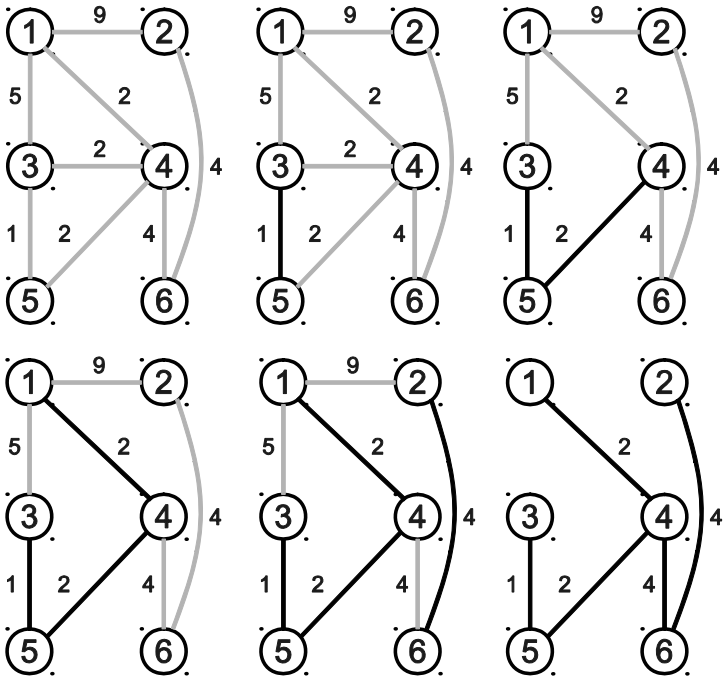
A Kruskal-algoritmikus a futás közben több fát bővít és von össze addig míg egy összefüggő feszítőfát nem kapunk, vagy már nincs lehetőség új él oly módon való beválasztására, hogy ezzel nem keletkezik kör. Az összevonások alapját az élek kiválasztása adja. Az algoritmus első lépésben növekvő sorrendbe rendezi a gráf éleit. A kapott élsorrendben haladva egy élt hozzáad a feszítőfához és az él által összekötött fákat összevonja, de csak akkor ha ezzel a lépéssel nem keletkezik kör. Az algoritmus pszeudó kódja a következő.

```
edges=a G=(V,E) gráf élei súly szerint növekvő sorrendben rendezve
tree=null;
for(i=1;i<=|E| and |tree|<|V|-1;i++) {
if az edges-beli e él nem alkot kört a többi tree beli éllel,
    then adjuk e-t a tree-hez
    távolítsuk el az e élt edges elejéről
}
```

Az algoritmus a tree halmazban gyűjti azokat az éleket, melyek a feszítőfához tartoznak. Az algoritmus addig fut, míg az élek el nem fogynak, vagy létrejött egy feszítő fa (N csúcsú fa éleinek száma N-1). Az algoritmus futási ideje $O(E \log E)$.

Vegyük a Dijkstra algoritmushoz bemutatott példagráfot. A Kruskal algoritmus első lépésben rendezést hajt végre a gráf éleire. Ezzel megkaphatjuk az élek következő sorrendjét: edges=((3, 5), (4, 5), (3, 4), (1, 4), (2, 6), (4, 6), (1, 3), (1, 9)). Egy él azonosítására használjuk a kezdőcsúcs és a végcsúcs azonosítóit. Természetesen, mivel több azonos súlyú él is szerepel a gráfban, ezért a rendező algoritmus jellegéből adódóan más sorrendet is kaphatunk.

Kruskal algoritmusa első lépésben kiválasztja az edges első helyén szereplő élt: (3, 5). Az él hozzáadható a feszítő fához, mivel nem jön vele létre kör a fában. A következő lépésben a (4, 5) azonosítókat tartalmazó élt kell megvizsgálni. Ez az él is hozzáfűzhető a feszítőfához, hiszen nem jön létre vele kör. Harmadik lépésben a (3, 4) élt választjuk ki. Azonban ezzel az éllel kör jönne létre a 3, 4 és 5 csúcsok között, ezért ezt nem választhatjuk hozzá a feszítőfához. További két él hozzáadásával kapjuk meg a feladat minimális feszítőfáját. Az algoritmus futásának lépéseit a 28. ábrán ismertetjük.



28. ábra: A Kruskal-algoritmus lépéseinek bemutatása mintafeladat segítségével

Jarnik–Prim-algoritmus

Ez az algoritmus is a minimális súlyú feszítőfát határozza meg. Az algoritmus nagyon hasonlít a Dijkstra-algoritmusnál bemutatott gráf végigjárás lépéseivel. Ugyanis a Jarnik–Prim-algoritmus egy kezdő élről (csúcsból) elindulva addig bővíti a fát, amíg egy minimális súlyú feszítőfát nem kapunk. Míg a Kruskal-algoritmus megengedi, hogy a feszítő fához választott élekből alkotott gráf ne legyen összefüggő (azaz vannak olyan csúcsok, melyek között nincs út), addig a Jarnik-Prim algoritmus a futása során végig egy összefüggő gráf (fa) bővítésével határozza meg a minimális súlyú feszítőfát.

```
edges=a G=(V,E) gráf élei súly szerint növekvő sorrendben rendezve
tree=null
for(i=1;|tree|<|V|-1;i++) {
    for(j=1;j<|edges|;j++) {
        if az edges-beli e él nem alkot kört a többi tree beli
            éllel, és illeszkedik valamely tree beli csúcshoz
        then {
            adjuk e-t a tree-hez
            break;
        }
    }
}
```

A Kruskal-algoritmusnál bemutatott példát tekintve az algoritmus futása során a (2, 6) azonosítókkal jelölt él beválasztása nem történhet meg, hiszen ez az él nem illeszkedik a feszítőfához. A minimális feszítő fa ehhez a példánál megegyezik a Kruskal algoritmus feszítőfájával, azonban a feladat függvényében előfordulhat, hogy a két algoritmus minimális feszítőfája különbözik (a feszítőfa súlya természetesen ugyanaz).

A Jarnik–Prim-algoritmus egy rendezéssel indul, ami általában költséges feladat, jelen esetben lényegesen rontja az algoritmus hatékonyságát. Az élek rendezéséhez O(E lg E) futási idő szükséges. Ha az újonnan hozzáadandó csúcs kiválasztására egy kulcs alapján felépített prioritásos sor segítségével oldanánk meg, ahol a kulcs értéke a kiválasztott csúcsot valamely fabéli csúccsal összekötő minimális él súlyával egyezne meg, akkor az előzetes rendezés elkerülésével hatékonyabb megvalósítást érhetünk el. Ebben az esetben az algoritmus futási ideje O(E lgV)-re csökkenthető.

Hash tábla

Listában, tömbben való keresés során az elemek kulcsait hasonlítjuk össze és az aktuálisan legjobb elem adatait tároljuk. Kereshetünk szekvenciálisan/lineárisan, amikor is egymás után vizsgáljuk az adatokat, míg meg nem találjuk a keresett értéket, vagy az adathalmaz végére nem értünk. Rendezett adatok esetén binárisan kereshetünk oly módon, hogy középen két részre bontva az adatokat a középső elem értéke alapján el tudjuk dönteni, hogy melyik részben kell a keresést folytatnunk. Ez az ötlet vezet a bináris keresőfához.

Az előző keresési stratégiáknál nem ismertük, hogy a tömbben hol tároltuk el a keresett értéket. Ha a keresett elem értéke alapján ki tudnánk számolni a tárolásának a helyét, akkor akár egy lépésben megtalálhatnánk a keresett elemet. Ehhez egy olyan h függvényt kell létrehozni, ami egy K értéket (kulcsot, *key*) – amely érték egész, valós, vagy akár egy sztring is lehet – egy tömbindexszé transzformál. Ezt a h-t hasító függvénynek (*hash function*) nevezzük. Ha a h függvény a különböző kulcsokat különböző tömbindexszé alakítja, akkor a h függvény tökéletes. Ilyen tökéletes függvényt nehéz készíteni, hiszen általában még a tárolandó kulcsok számát se ismerjük előre.

Ha a hasító függvény különböző kulcsokra ugyanazt a tömbindexet adja vissza, akkor ugyanazon a helyen kellene két különböző értéket tárolni. Ezt az esetet ütközésnek (*collision*) nevezzük. Egy hasító függvény annál jobb, minél kevesebb ütközést eredményez.

Hasító függvény

Tegyük fel, hogy n elemet akarunk m helyre elpakolni ($n \leq m$). Mivel ezt m^n-féle módon tehetjük meg, ezért ennyi különböző hasító függvény létezik. Ebben a fejezetben néhány elterjedt, bevált hasító függvényt ismertetünk.

Az ideális hasító függvényt mindig az adott feladatnak megfelelően kell kiválasztani. Egy ideális hasító függvény egyenletesen osztja szét a kulcsokat, kevés ütközést generál, könnyen kiszámolhatónak kell lennie.

A hasító függvénynek egy valós tömbindexet kell hogy visszaadjon. Az osztó módszerben a maradékos osztás (modulo) egy jó választás, hiszen biztosítja, hogy az eredmény az osztónál kisebb lesz. Az osztónak választhatjuk a tömb méretét. Ha nem ismerjük a kulcsok eloszlását, akkor a legjobb, ha a maradékos osztáshoz a tömb méreténél kisebb prím számot választunk.

Jó választás lehet a négyzetközép módszer. A négyzetközép módszer először a vizsgált kulcs értékének a négyzetét veszi, majd utána visszaadja ebből a középső k darab számjegyből álló számot.

Bizonyos esetekben a tárolandó kulcsok nem számok, hanem karakterek, vagy esetleg karakterláncok. Ekkor általában a karakterekhez számértékeket rendelhetünk, majd ezekkel az értékekkel végezhetünk műveleteket (pl. összeadás).

Kulcs ütközés

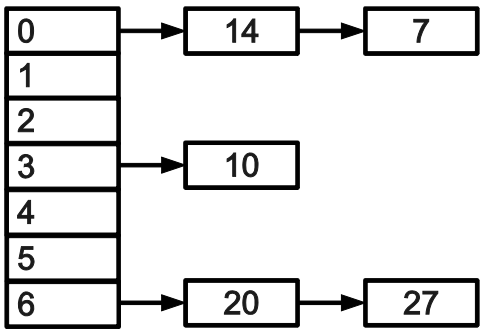
Kulcsütközésről akkor beszélünk, ha a h transzformáló függvény ugyanazzal az értékkel tér vissza különböző kulcsok esetén. Ez az eset könnyen előfordulhat, hiszen általában a tárolandó kulcsok száma jóval meghaladhatja a tárolásra használható tábla méretét.

Kulcsütközés esetén a legegyszerűbb megoldás lehet, hogy megpróbálunk egy olyan pozíciót találni, ahol még nincs érték tárolva. Ha a h(K) pozíció foglalt a táblában, akkor egy algoritmus alapján megpróbálunk új, üres pozíciót találni. A pozíció keresés egy próba függvény alapján a következő sorrendben történik: h(K)+p(i), h(K)+p(1), h(K)+p(2), …, h(K)+p(i), … . Ha a p(i)=i, akkor lineáris próbáról beszélünk. Azonban az ilyen próba függvények esetén is legfeljebb annyi kulcsot lehet a táblában tárolni, amennyi a tábla mérete.

Ha a tábla méreténél több kulcsot szeretnénk tárolni, vagy nem tudjuk előre hogy mennyi kulcs fog szerepelni a táblában, akkor a kulcsokat nem a közvetlenül a táblában kellene tárolni. Ha a tábla minden pozíciójához egy láncolt listát illesztünk, akkor ezekben a listákban tárolhatjuk a kulcsokat. A lista segítségével az ütközések feloldhatóak.

A láncolt listás hasító táblában a hasító függvény azt mondja meg, hogy új elem esetén melyik láncolt listába kell beszúrni azt. Kulcs keresése esetén a hasító függvény által jelölt láncolt listában kell ellenőrizni, hogy tartalmazza-e a keresett kulcsot.

A 29. ábrán egy hasító táblázatot láthatunk, melyben láncolt listában tároljuk a kulcsokat. A tábla hasító függvényének a héttel való maradékos osztást használjuk (h(K)=K mod 7). A héttel való maradékos osztás eredménye egy 0 és 6 közötti egész szám, ezekkel a számokkal indexeljük a táblát. Ha egy új kulcs érkezik, akkor azt a kulcshoz tartozó láncolt lista elejére szúrjuk be. Az ábrán látható hasító táblába a 10, 7, 27, 14, 20 kulcsokat tároltuk.



29. ábra: Példa láncolt listát használó hasító táblára

STL függvénykönyvtár

Előfordulhat, hogy egy adott osztályt, vagy adatszerkezetet több elemtípussal is meg akarunk valósítani (például egy olyan láncolt listát, amiben számokat vagy szöveget is tárolni szeretnénk). Ekkor lehetőségünk lenne létrehozni két különálló osztályt, egyet a számok tárolására alkalmas

láncolt lista, egyet pedig a szöveg tárolására alkalmas láncolt lista számára. Ennél a két listánál a különbség mindössze a tárolt érték típusában van, a láncolt lista többi része, és a lista műveletek ugyanazok lennének. A két különböző lista megvalósítása nagy méretű, felesleges kódismétléssel járna.

A kódismétlés kiküszöbölésére egy jó megoldás a behelyettesítő típusok, vagy sablonok (*template*) használata. A sablon használata esetén az osztály példányosítása esetén határozzuk meg, hogy mi az az adattípus, amit az adott objektumban használhatunk.

Az STL (*Standard Template Library*) egy C++ függvénykönyvtár a jegyzetben korábban bemutatott számos adatszerkezet tárolására. Az STL könyvtár sablonokat használ számos adatszerkezet és algoritmus megvalósítására. Az STL az adatszerkezet mellett tartalmazza az adatszerkezetekhez kapcsolódó fontosabb algoritmusokat és iterátorokat is. Az STL egy általános/generikus függvénykönyvtár, mely típus nélküli (template) osztály sablonokat tartalmaz. Az STL könyvtár előnye, hogy kellően általánosan lett létrehozva, bármely típusra hatékonyan alkalmazható. Az osztályok objektumorientált szemléletben kerültek implementálásra. STL adatszerkezetek robusztus megoldások. A különböző adatszerkezetekben használt metódusnevek a funkcióknak megfelelően megegyeznek. Az STL-en keresztül különböző beépített algoritmusokat érhetünk el.

Az STL számos tárolót, adatszerkezetet (container) tartalmaz. Azok az osztályok, melyeket STL-ben adatszerkezetek létrehozására használhatunk a következők: dinamikus tömb (vector), láncolt lista (list), sor (deque), halmazok (set, multiset), asszociatív tömbök (map, multimap) és különböző hash táblák (hash_set, hash_multiset, hash_map, hash_multimap). Ezek az osztályok mindegyike template típusú, azaz a deklaráció során a programozó döntheti el, hogy milyen adatokat kíván az osztályon belül tárolni. Például ha egészeket szeretnénk dinamikus tömbben tárolni, akkor a vector<int> típusra van szükségünk.

```
vector<int> v(2);
v[0] = 100;
v[1] = v[0] + 120;
```

Az STL az adatok tárolása mellett az adatok kezelésére többfajta lehetőséget és algoritmust biztosít. Például a reverse() függvény megfordítja a vector-ban tárolt elemek sorrendjét. A reverse függvény globális, nem a vector adatszerkezet tagfüggvénye. Azért nem tagfüggvényként hozták létre ezt a függvényt, hogy globálisan, más adatszerkezetek esetén is használni lehessen az adatszerkezetben tárolt elemek sorrendjének megfordítására. Az STL-beli adatszerkezeten túl ezt a függvényt használhatjuk akár egy C++ tömb elemeinek megfordítására is.

Az STL függvénykönyvtárban az adatszerkezet belső állapotainak tárolására és kezelésére iterátorokat (iterator) vezettek be. Az iterátorok ugyanolyan szerepet töltenek be, mint tömbök esetén mutatók. Az előbbi vector adatszerkezet esetén a v.begin() iterátor adja az adatszerkezet első elemét, a v.end() pedig az adatszerkezet utolsó elemét. A vector<int>::iterator típussal új iterátorokat hozhatunk létre, amivel az adatszerkezet elemeit elérhetjük. Az iterátorok segítségével saját algoritmusokat hozhatunk létre az STL adatszerkezetek kezelésére.

Összefoglalva: az STL könyvtárban találhatunk adatszerkezetektől független algoritmusokat, absztrakt adatszerkezeteket adatok tárolására és iterátorokat, melyeket adatszerkezetekben tárolt elemek elérésére használhatunk.

STL adatszerkezetek

Az STL olyan adatszerkezeteket (container) tartalmaz, melyben más elemeket gyűjthetünk össze. Ezeket az adatszerkezeteket sablon osztályonként implementálták, amivel nagyfokú rugalmasságot biztosít az ezt használó a programozó számára. Az adatszerkezet felelős az elemek tárolására szükséges memóriaterület kezeléséért és tagfüggvényeket biztosít a benne tárolt adatok eléréséhez. Ez az elérés megtörténhet direkt módon, vagy iterátorok segítségével.

Az STL könyvtárban megtalálhatjuk a legfontosabb adatszerkezeteket, amikre szükségünk lehet programozási feladataink során. Ilyen adatszerkezet a dinamikus tömb, a sor, a verem, a prioritásos sor, a láncolt lista, vagy az asszociatív tömb.

Az adatszerkezetek általában tartalmazzák azokat a tagfüggvényeket, melyekkel a hozzájuk kapcsolódó funkcionalitás implementálva lett. Annak eldöntése, hogy melyik adatszerkezetet válasszuk egy speciális programozási feladat megoldására nem csak az adatszerkezetben levő tagfüggvények funkcionalitásától függhet, hanem figyelembe kell venni a tagfüggvények számítási komplexitását is.

Az STL szekvenciális adatok tárolására három adatszerkezetet tartalmaz. Ezek az adatszerkezetek a dinamikus tömb (vector), a láncolt lista (list) és a kétvégű sor (deque). A verem (stack, LIFO), sor (queue, FIFO) és prioritásos sor (priority_queue) a korábbi adatszerkezetekre épülve valósítanak meg egy új adatszerkezetet. A halmaz (set, bitset), multihalmaz (multiset), asszociatív tömb (map, multimap) az asszociatív tárolók közé tartozik.

Az STL könyvtárban bizonyos adatszerkezetek eléréséhez biztosítanak indexelést (például: vector, map), azonban egyes típusok indexeléssel nem járhatóak be, hanem csak külön bejárásra biztosított objektumokkal (iterator) járhatjuk végig a benne tárolt adatokat.

Láncolt lista az STL-ben

A láncolt lista adatszerkezet egy duplán láncolt lista formájában került implementálásra, mely mutatókat tartalmaz a lista fejére és a végére. Ezt a tárolót csak akkor használhatjuk, ha a program elején az #include <list> direktívával betöltöttük.

Az STL könyvtárban a láncolt lista a list adatszerkezetben került megvalósításra. A list szekvenciális adatszerkezet, mely számos lehetőséget biztosít a láncolt lista használata során. Lehetőséget ad bármely elemének a módosítására, törlésére (akár egymás után több elemet is), bármely elem elé, vagy mögé beszúrhatunk egy vagy több új elemet. Kicserélhetünk adott részeket újra a listán belül. Az elemek sorrendjét változtathatjuk, rendezhetjük a listát ha akarjuk.

A láncolt listában számos tagfüggvényt találhatunk az adatok manipulálására és elérésére. A teljesség igénye nélkül néhány tagfüggvény:

- a lista tartalmaz különböző konstruktorokat, destruktort és másoló konstruktort
- begin() – iterátor ami a lista első elemét adja vissza
- end() – iterátor ami a lista utolsó elemét adja vissza
- empty() – függvény megmondja, hogy a lista üres-e
- size() – visszaadja a listában tárolt elemek számát
- max_size() – visszaadja a lista maximális elemszámát
- assign() – törli a lista elemeit, majd új elemekkel tölti fel
- push_front() – új elemet ad hozzá a lista elejére

- pop_front() – kiveszi a lista elején található első elemet
- push_back() – új elemet ad hozzá a lista végére
- pop_back() – kiveszi a lista végén található első elemet
- insert() – lista elemek beszúrása az adatszerkezetbe
- erase() – elemek törlése
- clear() – lista tartalmának törlése
- splice() – egyik listából a másikba lehet vele elemeket átrakodni
- unique() – olyan elemeket töröl a listából, melyek többször szerepelnek benne
- sort() – rendezés
- reverse() – elemek fordított sorrendbe állítása

Mivel a memóriában egy láncolt lista szerkezetben kerülnek letárolásra a list elemei, ezért nem járhatóak be az elemeik közvetlen indexeléssel. A bejárásra a list::iterator adattípust használhatjuk, melyre működik a léptetés operátor.

Az STL könyvtár vector adatszerkezete

Az STL vector adatszerkezete dinamikus tömb adatszerkezetet valósít meg. Ez gyakorlatilag egy folytonos memóriaterület, amit adattárolásra használhatunk. A vector alkalmazása esetén a konténer maga gondoskodik a memória lefoglalásáról és felszabadításáról.

A vector adatszerkezet a hagyományos C/C++-beli tömbök helyett tervezték. A vector nagy előnye, hogy mind a statikus, mind a dinamikus tömb kezeléséhez képest önmaga gondoskodik a a dinamikus méret változtatásról.

Az STL könyvtár Queue adatszerkezete

A sor egy olyan FIFO adatszerkezet, melynek végére tudunk betenni elemet és az elejéről tudunk kivenni benne levő értéket. Az STL könyvtárban a queue osztály valósítja meg a sor adatszerkezetet.

A queue osztály a következő metódusokkal használható:

- push() – érték behelyezése a sor végére
- pop() – érték kivétele a sor elejéről
- front() – az első elem lekérdezése
- back() – az utolsó elem lekérdezése
- size() – a sorban tárolt értékek számának lekérdezése
- empty() – a sor ürességének a lekérdezése

Nézzünk egy rövid példát a sor adatszerkezet használatára. Tegyünk be a sorba három egész számot, majd vegyük ki a sor elején található értéket:

```
queue<int> intQueue;


intQueue.push(-33);
intQueue.push(43);
intQueue.push(7);
```

```
cout << intQueue.front() << endl; //a kiírt érték: -33
intQueue.pop();
cout << intQueue.front() << endl; //a kiírt érték: 43
```

Az STL könyvtár Deque adatszerkezete

A kétvégű sor egy olyan láncolt lista, mely mindkét végére tudunk elemeket berakni és kivenni. Ezt az adatszerkezetet egy kétszeresen láncolt lista segítségével implementálták ugyanúgy, mint a list adatszerkezetet. Ahogy már a list-ben található funkcionalitás mellett a kétvégű sornál a közvetlen címzést is beleépítették (úgy mint a vectornál is). Míg a vectornál nehézkes az új elem beszúrása, mert ez adatok átmozgatásával jár, itt a beszúrás kevesebb memóriaművelettel jár a láncolt lista miatt.

Az STL verem adatszerkezete

A verem (*stack*) egy olyan adatszerkezet, melynek tetejére tudunk új elemet behelyezni, illetve csak a tetején szereplő elemet tudjuk elérni és kivenni. A mélyebben levő elemek rejtettek a külvilág számára. A mérete növelhető tetszőlegesen, azonban általában lehetőség van a méret korlátozására (szükség esetén).

Az STL könyvtárban a verem típust a stack osztály valósítja meg. A stack osztály legfontosabb műveletei:

- push() – érték berakása a verem tetejére
- pop() – érték kivétele a verem tetejéről
- top() – a verem tetején levő érték lekérdezése
- empty() – a verem ürességének a lekérdezése
- size() – a verem méretének a lekérdezése

Nézzünk egy rövid példát a verem használatára. Tegyünk be a verembe három egész számot, majd vegyük ki a verem tetején található értéket:

```
stack<int> intStack;

intStack.push(-33);
intStack.push(43);
intStack.push(7);

cout << intStack.top() << endl; //a kiírt érték: 7
intStack.pop();
cout << intStack.top() << endl; //a kiírt érték: 43
```

Verem implementálása STL vector-al

A verem osztály megvalósításának egyik legegyszerűbb módja, ha egy tömböt használunk a veremben szereplő elemek tárolására. A tömb kezelését (lefoglalás, felszabadítás, hozzáférés, ellenőrzések) megvalósíthatjuk mi is, azonban igénybe vehetjük az STL-ben található vector tömböt.

Egy általános verem osztály a következőképpen deklarálható az STL vector adatszerkezet segítségével:

```
#include <vector>

template<class T, int capacity>
class myStack {
public:
    myStack() {myVector.reserve(capacity);}
    void clear() {myVector.clear();}
    bool isEmpty() {myVector.empty();}
    T pop() const {
        T element = myVector.back();
        myVector.pop_back();
        return element;
    }
    void push(const T& element) {myVector.push_back(element);}
private:
    vector<T> myVector;
}
```

Az STL könyvtár map adatszerkezete

Az STL könyvtárban létrehoztak egy asszociatív tömböt, ami egy olyan vektor mely indexelése nem csak az egész számokkal történhet, hanem tetszőleges típusú indexet használhatunk erre. Az asszociatív tömb esetén az indexeket szokás kulcsoknak is nevezni. Az asszociatív tömb esetén feltétel, hogy két ugyanolyan kulcs nem szerepelhet benne. Az asszociatív tömbben az elemeknek nincs rögzített sorrendje.

Az STL-ben az asszociatív tömböt a map osztály implementálja. Az osztály legfontosabb metódusai, műveletei:

- []: kulcs lekérdezése
- find() – kulcs keresése
- insert() – elem beszúrás
- erase() – elem törlés
- swap() – elemek cseréje
- clear() – ürítés
- empty() – üresség lekérdezése
- size() – méret lekérdezése

Az elemek bejárására itt is egy iterátort kell használni (map::iterator), melynek két attribútuma van: a kulcs a first, az érték a second attribútumban szerepel.

Algoritmusok az STL-ben

Az STL algoritmusok az <algorithm> csatolásával érhetőek el. Az algoritmusok között az alap informatikai algoritmusokat találhatjuk meg (másolás, feltöltés, keresés, bináris keresés, rendezések).

Az STL többfajta rendező algoritmust is biztosít adathalmazok rendezésére, például gyorsrendezést, kupac rendezést, vagy összefésülő rendezést. A rendezések gazdagon paraméterezhetőek, megadható hogy milyen tartományban milyen szempontok alapján történjen a rendezés.

Az algoritmusok között vannak olyanok, melyek bejáró típusúak, vannak rendező algoritmusok, sorozatmódosító algoritmusok és egy speciális algoritmusok. Az STL konténerek feltételezik, hogy vannak függvények a paraméterül adott típusokra (például az operátorokra). Ezért bonyolult algoritmusokat hoztak létre a konténer típusának rögzítése nélkül.

Tekintsünk át az algoritmusok közül néhány fontosabbat:

- Bejáró típus: for_each, find, count, ...
- Rendezés: sort, stable_sort,...
- Módosító: swap, fill, replace,...
- Egyéb: binary_search, sort_heap,...

A legfontosabb algoritmusok a következők:

- for_each: minden elemre végrehajt egy paraméterként kapott függvényt
- find: megkeres egy elemet, ha nincs benne a konténerben, akkor az end()-el tér vissza
- find_if: megkeresi az első olyan elemet, amire a paraméterként kapott feltétel teljesül
- count: megszámolja, hogy egy érték hányszor fordul elő
- copy: elemek másolása
- fill: tároló feltöltése egy adott elemmel
- unique: csak egy érték maradhat egymás utáni azonos elemekből
- reverse: megfordítja az intervallumot
- sort: növekvő sorrendbe rendezés
- stable_sort: növekvő sorrendbe rendezés (stabil rendezés)
- min_element: legkisebb elem iterátora

Iterátorok az STL könyvtárban

Az iterátor egy olyan osztály, mely segítségével egy adatszerkezet elemeit elérhetjük. Azzal, hogy az STL könyvtárban az iterátorokat önmagukban hozták létre, lehetővé vált az algoritmusok és az adatszerkezetek függetlenítése.

Az iterátor programozástechnológia szempontjából a mutatók absztrakciójának felelnek meg. Az iterátor a tároló osztályban definiált lokális osztály. Az iterátorokkal műveleteket végezhetünk, olvashatjuk (=*p) a tartalmát, módosíthatjuk a tartalmát (*p=). Definiálva van rajtuk a léptető operátor (++, --, +=). Ha szükséges összehasonlíthatjuk iterátorok egyenlőségét (==,!=, < <=).

A következő példában egy list adatszerkezetet járunk be egy iterátor segítségével. A példában a szavak adatszerkezetben sztringeket tárolunk láncolt lista segítségével. Az it iterátort használjuk a láncolt lista végigjárására:

```
std::list<std::string> szavak;
std::list<std::string>::iterator it;
for (it=szavak.begin(); it!=szavak.end(); ++it) {
    std::cout<<*it<<' ';
}
```

Felhasznált szakirodalom

Dr. Kondorosi Károly, Dr. László Zoltán, Dr. Szirma-Kalos László: Objektum-Orientált Szoftverfejlesztés, Computerbooks 1999.

Ian Sommerville: Szoftverrendszerek fejlesztése, Panem Kiadó, 2007

T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein: Introduction to Algorithms, Second Edition

A. Drozdek: Data Structures and Algorithms in C++, Brooks and Cole, 2001

Rónyai – Ivanyos – Szabó: Algoritmusok, Typotex 1999

Melléklet

Láncolt lista megvalósítása C++ programozási nyelven

```
#include <iostream>
using namespace std;

class IntNode{
public:
    IntNode() {
        next = 0;
    }
    IntNode(int i, IntNode *new_node = 0) {
        data = i; next = new_node;
    }
    int data;
    IntNode *next;
};

class IntList {
private:
    IntNode *head, *tail;
public:
    IntList() {head = tail = 0;}
    ~IntList();
    bool isEmpty() {return head == 0;}
    void addToHead(int);
    void addToTail(int);
    int deleteFromHead();
    int deleteFromTail();

};

IntList::~IntList() {
            IntNode *temp;
            while(!isEmpty()){
                        temp = head->next;
                        delete head;
                        head = temp;
            }
}

void IntList::addToHead(int new_data) {
    head = new IntNode(new_data, head);
    if(tail == 0)
        tail = head;
}

void IntList::addToTail(int new_data) {
    if(tail != 0) {
        tail->next = new IntNode(new_data);
        tail = tail->next;
    } else
    head = tail = new IntNode(new_data);
}

int IntList::deleteFromHead() {
            if(head) {
                        int value = head->data;
                        IntNode *temp = head;
                        if(head == tail)
                                    head = tail = 0;
                        else
                                    head = head->next;
                        delete temp;
                        return value;
            } else
                        return 0;
}
```

```
int IntList::deleteFromTail() {
        if(tail){
                int value = tail->data;
                if(head == tail) {
                        delete tail;
                        head = tail = 0;
                }
                else {
                        IntNode *temp;
                        for(temp=head; temp->next != tail; temp = temp->next)
                                ;
                        delete tail;
                        tail = temp;
                        tail->next = 0;
                }
                return value;
        } else
                return 0;
}

int main() {
        IntList L;

        L.addToHead(10);
        L.addToHead(9);
        L.addToHead(8);
        L.addToHead(7);
        L.addToHead(6);

        cout << L.deleteFromTail() << endl;
        cout << L.deleteFromTail() << endl;
        cout << L.deleteFromHead() << endl;
        cout << L.deleteFromHead() << endl;
        cout << L.deleteFromHead() << endl;
        cout << L.deleteFromTail() << endl;

        cout << "END" << endl;
        return 1;
}
```

Verem megvalósítása C++ programozási nyelven tömb segítségével

```
#include <iostream>
using namespace std;

class IntStack {
public:
    IntStack(int cap) {
        top = 0; capacity = cap;
        data = new int[capacity];
    }
    ~IntStack() {
            delete [] data;
    }
    bool isEmpty() const {return top==0;}
    void Push(int i);
    int Pop();
    void Clear();
private:
    int *data;
    int capacity;
    int top;
};

int IntStack::Pop() {
    if(top > 0) {
        top--;
        return data[top];
```

```
    } else {
            cout <<"Stack is empty" << endl; //ures verem hibauzenet
            return -1;
    }
}

void IntStack::Push(int i) {
    if(top < capacity) {
        data[top] = i;
        top++;
    } else cout << "Stack is full" << endl; //tele a verem hibaÄľzenet
}

void IntStack::Clear() {
    top = 0;
}

int main() {
        IntStack S(5);

        int d;

        d = S.Pop();
        S.Push(7);
        S.Push(3);
        S.Push(0);
        S.Push(6);
        S.Push(10);
        S.Push(9);

        return 1;
}
```

Sor megvalósítása C++ nyelven tömb segítségével

```
#include <iostream>
using namespace std;

#define MAX 5

class IntQueue {
private:
    int data[MAX];
    int beginI, endI;
public:
    IntQueue() {
        beginI = endI = -1;
    }
    bool isEmpty() const;
    int dequeue();
    void enqueue(int);
};

bool IntQueue::isEmpty() const {
        if(endI == -1)
                return true;
        else
                return false;
}

int IntQueue::dequeue() {
    int tmp = -1;
    if(endI==-1)
        cout << "Queue is empty" << endl;
    else{
        tmp = data[beginI];
        if(beginI==endI)
            beginI = endI = -1;
        else
            beginI = ++beginI % MAX;
    }
```

```
    return tmp;
}

void IntQueue::enqueue(int tmp) {
   if(endI == -1){
        beginI = endI = 0;
        data[0] = tmp;
   } else {
        if((endI+1) % MAX == beginI)
            cout << "Queue is full"<< endl;
        else {
            endI = ++endI % MAX;
            data[endI] = tmp;
        }
   }
}

int main() {
            IntQueue Q;
            int d;

            d = Q.dequeue();
            Q.enqueue(4);
            Q.enqueue(5);
            Q.enqueue(8);
            Q.enqueue(1);
            Q.enqueue(2);
            Q.enqueue(12);

            cout << "END" << endl;
            return 0;
}
```

Bináris fa szélességi bejárása

Main.cpp

```
#include "IntTreeNodeQueue.hpp"

int countIntNodes(IntTreeNode *root) {
    if ( root == NULL )
        return 0;
    else {
        int count = 1;
        count += countIntNodes(root->left);
        count += countIntNodes(root->right);
        return count;
    }
}

void BFSIntTree(IntTreeNode *root) {
    IntTreeNodeQueue queue;
    IntTreeNode *p;
    if(root != 0)
        queue.enqueue(root);
    while(!queue.isEmpty()) {
        p = queue.dequeue();
        cout << p->data << " "; //visit p
        if(p->left != 0)
            queue.enqueue(p->left);
        if(p->right != 0)
            queue.enqueue(p->right);
    }
}

int main() {
            IntTreeNode *root;

            root = new IntTreeNode;
            root->data = 43;

            root->left = new IntTreeNode;
            root->left->data = 1;
            root->left->left = NULL;
            root->left->right = NULL;

            root->right = new IntTreeNode;
            root->right->data = 3;
            root->right->left = NULL;
            root->right->right = NULL;

            cout << countIntNodes(root) << endl;

            BFSIntTree(root);

            //delete root !!!
            return 0;
}
```

IntTreeNodeQueue.hpp

```
#ifndef INTTREENODEQUEUE_HPP_
#define INTTREENODEQUEUE_HPP_
#include <iostream>

using namespace std;

#define MAX 100

struct IntTreeNode {
    int data;
    IntTreeNode *left;
    IntTreeNode *right;
};

class IntTreeNodeQueue {
private:
    IntTreeNode* data[MAX];
    int beginI, endI;
public:
    IntTreeNodeQueue() {
        beginI = endI = -1;
    }
    bool isEmpty() const;
    IntTreeNode* dequeue();
    void enqueue(IntTreeNode*);
};

bool IntTreeNodeQueue::isEmpty() const {
            if(endI == -1)
                        return true;
            else
                        return false;
}

IntTreeNode* IntTreeNodeQueue::dequeue() {
    IntTreeNode* tmp = NULL;
    if(endI==-1)
        cout << "Queue is empty" << endl;
    else{
        tmp = data[beginI];
        if(beginI==endI)
            beginI = endI = -1;
        else
            beginI = ++beginI % MAX;
    }
    return tmp;
}

void IntTreeNodeQueue::enqueue(IntTreeNode* tmp) {
   if(endI == -1){
        beginI = endI = 0;
        data[0] = tmp;
   } else {
        if((endI+1) % MAX == beginI)
            cout << "Queue is full"<< endl;
        else {
            endI = ++endI % MAX;
            data[endI] = tmp;
        }
   }
}
#endif /* INTTREENODEQUEUE_HPP_ */
```

Bináris kereső fa megvalósítása

```
#include <iostream>
using namespace std;

class BSTIntNode{
public:
        BSTIntNode() {
                left = right = 0;
        }
        int key;
        BSTIntNode *left, *right;
};

class BST {
public:
        BST() { root = 0; }
        BST(int);
        ~BST() {/*delete root rekurzivan*/}
        bool isEmpty() const { return root==0; }
        int searchTree(const int element) {
                return search(root, element);
        }
        int minimum(BSTIntNode *) const;
        int maximum(BSTIntNode *) const;
protected:
        int search(BSTIntNode*, const int) const;
private:
        BSTIntNode *root;
};

int BST::search(BSTIntNode *p, const int element) const {
        while(p != 0) {
                if(element == p->key)
                        return p->key;
                else if(element < p->key)
                        p = p->left;
        else
                p = p->right;
        }
        return 0;
}

int BST::minimum(BSTIntNode *p = NULL) const {
        if(!p) p = root;
        while(p->left != 0) {
                p = p->left;
        }
        return p->key;
}

int BST::maximum(BSTIntNode *p = NULL) const {
        if(!p) p = root;
        while(p->right != 0) {
                p = p->right;
        }
        return p->key;
}

BST::BST(int n) {
        root = new BSTIntNode;
        BSTIntNode *p1 = new BSTIntNode;
        BSTIntNode *p2 = new BSTIntNode;
        BSTIntNode *p3 = new BSTIntNode;
        BSTIntNode *p4 = new BSTIntNode;
        BSTIntNode *p5 = new BSTIntNode;
        BSTIntNode *p6 = new BSTIntNode;
        BSTIntNode *p7 = new BSTIntNode;

        root->key = 13; root->left = p1; root->right = p2;
        p1->key = 10; p1->left = p3; p1->right = p4;
        p2->key = 25; p2->left = p5; p2->right = p6;
```

```
        p3->key =  4; p3->left = NULL; p3->right = NULL;
        p4->key = 12; p4->left = NULL; p4->right = NULL;
        p5->key = 20; p5->left = NULL; p5->right = NULL;
        p6->key = 31; p6->left = p7; p6->right = NULL;
        p7->key = 29; p7->left = NULL; p7->right = NULL;
}

int main() {
        BST T(1);

        cout << "Maximum:" << T.maximum() << endl;
        cout << "Minimum:" << T.minimum() << endl;
        cout << "Kereses( 7):" << T.searchTree(7) << endl;
        cout << "Kereses(10):" << T.searchTree(10) << endl;

        return 0;
}
```

Gráf mélységi és szélességi bejárása

Az IntStack.hpp és az IntQueue.hpp fájlok a korábban bemutatott IntStack és IntQueue adatszerkezeteket tartalmazza.

```
#include "IntStack.hpp"
#include "IntQueue.hpp"

int ADJ[5][5] = {{0,1,1,0,0},
                 {1,0,0,1,0},
                 {1,0,0,1,1},
                 {0,1,1,0,0},
                 {0,0,1,0,0}};
bool visited[5] = {false,false,false,false,false};
int d[5] = {0,0,0,0,0};
int N = 5;

void DFS(int start_node) {
        int i,j,step;
        IntStack S(100);
        step=0;
        S.Push(start_node);
        while(!S.isEmpty()) {
                i=S.Pop();
                if(visited[i]==false){
                        visited[i]=true;
                        d[i]=step++;
                        for(j=0;j<N;j++){
                                if(ADJ[i][j]==1 && visited[j]==false)
                                S.Push(j);
                        }
                }
        }
}

void BFS(int start_node) {
        IntQueue Q;
        int i,j,step;
        step=0;
        Q.enqueue(start_node);
        while(!Q.isEmpty()) {
            i=Q.dequeue();
            if(visited[i]==false){
                visited[i]=true;
                d[i]=step++;
                for(j=0;j<N;j++){
                    if(ADJ[i][j]==1 && visited[j]==false)
                        Q.enqueue(j);
                }
            }
        }
}

int main() {
        int i;

        DFS(0);
        cout << "DFS sorrend:";
        for(i = 0;i < N; i++)
                cout << d[i] << " ";
        cout << endl;

        for(i = 0;i < N;visited[i++] = false);

        BFS(0);
        cout << "BFS sorrend:";
        for(i = 0;i < N; i++)
                cout << d[i] << " ";


        return 0;
}
```

Dijkstra algoritmusa

```
#include <iostream>
using namespace std;
#define INF 10000

int ADJ[6][6]= {{INF,  9,  5,  2,INF,INF},
                                        {  9,INF,INF,INF,INF,  4},
                                        {  5,INF,INF,  2,  1,INF},
                                        {  2,INF,  2,INF,  2,  4},
                                        {INF,INF,  1,  2,INF,INF},
                                        {INF,  4,INF,  4,INF,INF}};
int dist[6]={INF,INF,INF,INF,INF,INF};
bool fix[6] = {false,false,false,false,false,false};
int pred[6] = {-1,-1,-1,-1,-1,-1};
int N = 6;

int GetClosestNode() {
        int i, closest = -1;
        for(i = 0;i < N; i++){
                if(!fix[i] && dist[i] < INF){
                        if(closest == -1)
                                closest = i;
                        else if(dist[i]<dist[closest])
                                closest = i;
                }
        }
        return closest;
}

void Dijkstra(int start_node){
        int i, j, closest;
        dist[start_node]=0;
        for(i = 0;i < N; i++) {
                closest=GetClosestNode();
                if(closest == -1) break;
                fix[closest] = true;
                for(j = 0;j < N;j++){
                        if(dist[j]>dist[closest]+ADJ[closest][j]){
                                dist[j]=dist[closest]+ADJ[closest][j];
                                pred[j]=closest;
                        }
                }
        }
}

int main() {
        int i;
        Dijkstra(2);
        cout << "Legrovidebb utak a 2 csucsbol:";
        for(i  = 0;i < N;i++)
                cout << dist[i] << " ";
        cout << endl;
        cout << "Elozo csucsok:";
                for(i  = 0;i < N;i++)
                        cout << pred[i] << " ";
        return 0;
}
```

Bellman-Ford algoritmusa szomszédsági mátrixszal

```
#include <iostream>
using namespace std;

#define INF 10000

int ADJ[6][6]= {{INF,  9,  5,  2,INF,INF},
                                        {  9,INF,INF,INF,INF,  4},
                                        {  5,INF,INF,  2, -1,INF},
                                        {  2,INF,  2,INF,  2,  4},
```

```
                                        {INF,INF,  3,  2,INF,INF},
                                        {INF,  4,INF,  4,INF,INF}};
int dist[6]={INF,INF,INF,INF,INF,INF};
int pred[6] = {-1,-1,-1,-1,-1,-1};
int N = 6;

void BellmanFord(int start_node){
        int i, j, k;
        dist[start_node]=0;
        for(i = 0;i < N-1; i++)
                for(j = 0;j < N;j++)
                        for(k = 0;k < N;k++)
                                if(ADJ[j][k]< INF)
                                        if(dist[k]>dist[j]+ADJ[j][k]) {
                                                dist[k]=dist[j]+ADJ[j][k];
                                                pred[k]=j;
                                        }
}

int main() {
        int i;
        BellmanFord(2);
        cout << "Legrovidebb utak a 2 csucsbol:";
        for(i  = 0;i < N;i++)
                cout << dist[i] << " ";
        cout << endl;
        cout << "Elozo csucsok:";
                for(i  = 0;i < N;i++)
                        cout << pred[i] << " ";
        return 0;
}
```