Eötvös Loránd Tudományegyetem
Informatikai Kar

Kupac adatszerkezetek implementálása és összehasonlítása

Készítette: Batha Dorián (BADNAAI.ELTE)
Programtervező informatikus (BSc) szak, nappali tagozat
Témavezető: Dezső Balázs

Budapest, 2008. június

Tartalomjegyzék

Előszó .... 4. oldal
1. **Bevezető** .... 5. oldal
   1.1. Gráfelméleti alapfogalmak .... 5. oldal
   1.2. A gráfok ábrázolási módszerei .... 5. oldal
2. **Felhasználói dokumentáció** .... 8. oldal
   2.1. Adott csúcsból induló legrövidebb utak problémája .... 8. oldal
      2.1.1. Dijkstra-algoritmus .... 10. oldal
   2.2. A LEMON könyvtár .... 12. oldal
3. **Fejlesztői dokumentáció** .... 14. oldal
   3.1. Kupac-implementációk .... 14. oldal
      3.1.1. Bináris, 4-gyerekes, K-gyerekes kupac .... 15. oldal
      3.1.2. Binomiális kupac .... 20. oldal
      3.1.3. Fibonacci kupac .... 26. oldal
      3.1.4. Párosítós kupac .... 28. oldal
   3.2. Tesztelés .... 35. oldal
      3.2.1. A K-gyerekes kupac tesztelése K = 4,16,32 esetekben .... 36. oldal
      3.2.2. A bináris és a K=16-gyerekes kupac összehasonlítása .... 37. oldal
      3.2.3. A 4-gyerekes és a K=4-gyerekes kupacok összehasonlítása .... 38. oldal
      3.2.4. A binomiális, párosítós és a Fibonacci kupac tesztelése .... 39. oldal
      3.2.5. New York úthálózatán futtatott Dijkstra-algoritmusok .... 39. oldal
      3.2.6. Értékelés .... 40. oldal
4. **Összefoglalás** .... 41. oldal
Irodalomjegyzék .... 42. oldal
Köszönetnyilvánítás .... 43. oldal

Előszó

„A tudomány tényekből építkezik, ahogy egy ház kövekből. Tények együttese azonban még ugyanúgy nem tudomány, ahogy egy halom kő sem ház.”

Henri Poincaré

A Dijkstra-algoritmus nagyon sok programnak fontos építőköve. Hatékony és gyors futása sok probléma megoldását egyszerűsítheti, illetve gyorsíthatja fel. Ezért választottam szakdolgozatom témájának, hisz nagyon sok gyakorlati feladat megoldásában, program felépítésében ez az algoritmus az alapkő. Ilyenek például a minimális költségű folyam, súlyozott párosítás páros gráfban, T-join és a több termékes folyam. A LEMON, mely egy C++ könyvtár nagy segítségét nyújt gráfok reprezentálásában, gráfokkal kapcsolatos problémákat megoldó algoritmusok hatékony és egyszerűbb implementációjában.

A szakdolgozatomat az elméleti alapoktól kiindulva építem fel egészen a Dijkstra-algoritmus bemutatásáig. Először egy kis bevezetőben a legfontosabb gráfelméleti fogalmakat gyűjtöm össze rendszerezve, majd a gráfok főbb tárolási módszereit ismertetem. Ezután kerül sorra a legrövidebb utak problémája és annak egyik megoldása a Dijkstra-algoritmus. Majd következik a szakdolgozat fő témája a Dijkstra-algoritmus által használt rendezett gráf tárolási módszerek, a különböző típusú, fajtájú kupacok. Megvalósítom a bináris mintájára a 4-gyerekes és az általános K-gyerekes kupacot. Ezen kívül még a binomiális és a párosítós kupac is implementálásra kerül. Végül ezeket tesztelem különböző méretű és sűrűségű gráfokon.

Bevezető

1.1. Gráfelméleti alapfogalmak

A halmaz, különböző objektumok együttese, amelyeket az illető halmaz elemeinek nevezzük. Ha az A halmaz minden eleme a B halmaznak is eleme, akkor azt mondjuk, hogy A részhalmaza B-nek. Két halmaz (A és B) Descartes-szorzata az összes olyan rendezett párból álló halmaz, amelynek az első komponense A-ban, második B-ben van. Az A, B halmazok AxB Descartes-szorzatának valamely R részhalmazát, az A, B halmazokon értelmezett (bináris) relációnak nevezzük.

A gráfoknak két típusa van az irányított és az irányítatlan. Irányított gráfon egy $G = (V, E)$ rendezett párt értünk, ahol V egy véges halmaz, E pedig egy bináris reláció V-n. V-t a G-beli csúcsok halmazának, az elemeit csúcsoknak nevezzük. Az E a G éleinek a halmaza, az E-beli elemeket éleknek nevezzük. Legyen (u, v) egy él a $G = (V, E)$ gráfban, ekkor a v csúcsot az u csúcs szomszédjának nevezzük. Ha a gráf irányított, akkor a szomszédsági reláció nem feltétlen tesz eleget a szimmetriának. A G gráfban az u csúcsot az u'-vel összekötő k hosszúságú úton csúcsoknak egy olyan (véges) $< v_0, v_1, \dots, v_k >$ sorozatát értjük, amelyre $u = v_0$, $u' = v_k$ és $(v_{i-1}, v_i) \in E$, $(i = 1,2 \dots k)$. Az út hossza az útban szereplő élek száma. Egy irányított gráfban a $< v_0, v_1, \dots, v_k >$ út kört alkot, ha $v_0 = v_k$ és a szóban forgó út tartalmaz legalább egy élt. A hurok egy 1 hosszúságú kör. A gráf körmentes, ha nem tartalmaz kört.

1.2. A gráfok ábrázolási módszerei

A gráfok tárolásánál alapvetően kétféle adattípust használnak. Az egyik, a tömbös megvalósítás, azaz szomszédsági mátrixban tárolják a gráfot. Előnye, hogy gyorsan lehet keresni, beszúrni, módosítani és törölni. Hátránya ugyanakkor, hogy előfordulhat, hogy túl nagy mátrixokat kapunk, amelyek tárolása nehézkes, hiszen nagy memória és merevlemezigényre van szükség. Ezért ezt a megvalósítási módot ritka gráfok esetében szokták alkalmazni.

A másik a láncolt listás megvalósítás, tehát ekkor szomszédsági listával ábrázolják a gráfot. Ennek előnye, hogy nagyméretű gráfokat is viszonylag egyszerűen tudunk tárolni, kezelni, ezért ezt sűrű gráfok esetén használják. Például a Dijkstra-

algoritmus esetén feltételezzük, hogy a G gráf egy szomszédsági listával van megadva. A láncolt lista egyszerű és rugalmas eszköz a dinamikus halmazok ábrázolására, és lehetővé teszi azok műveletinek megvalósítását. Ezek a dinamikus halmazok bővülhetnek, zsugorodhatnak, vagy másképpen módosulhatnak időben. A láncolt lista olyan adatszerkezet, amelyben az objektumok lineáris sorrendben követik egymást. A tömbök esetén a lineáris sorrendet az indexek határozzák meg, míg ezzel szemben a listákban a mutatók töltik be ezt a szerepet. Minden elem tartalmaz egy mutatót, amely a következő elemre mutat. A lista adatszerkezetének számos változata van. Lehet egyszeresen vagy kétszeresen láncolt, rendezett vagy nem rendezett, és lehet ciklikus vagy nem ciklikus is. Ha egy lista egyszeresen láncolt, akkor elemeiben csak a $köv$ mutató szerepel, mely értelemszerűen a következő elemre mutat. Ha rendezett, akkor a mutatók által meghatározott sorrend megegyezik az elemekben tárolt kulcsmezők növekvő sorrendjével, tehát a legkisebb kulcsérték a lista fejében, a legnagyobb pedig a lista végében található, azaz az utolsó elemében szerepel. Ha egy lista rendezetlen, akkor eleminek sorrendje tetszőleges lehet. A ciklikus listát az jellemzi, hogy a fej $előző$ mutatója a lista végére, a vége $köv$ pointere pedig a fejre mutat. A ciklikus listát ezért úgyis tekinthetjük, mint egy elemekből alkotott gyűrűt. A korábban említett mindkét megvalósítási módszer esetében létezik a keresés, beszúrás, törlés művelet.

Továbbiakban a láncolt listás megvalósítás esetében definiálom a fenti műveleteket. A keresés lineáris módon megkeresi az első k kulcsú elemet az L listában, és az arra mutató pointert adja vissza, ha megtalálja. Ha nincs k kulcsú elem a listában, akkor a visszaadott érték NIL. Ha egy n elemű listában keresünk, akkor lehetséges, hogy miden elemet meg kell vizsgálni, ezért a LISTÁBAN-KERES eljárás végrehajtási ideje a legrosszabb esetben $\Theta(\mathrm{n})$. A LISTÁBA-BESZÚR(L, x) egy megadott x elemet „befűzi" a lista elejére. Egy n elemű listára végrehajtott LISTÁBA-BESZÚR művelet futási ideje $O(1)$. A LISTÁBÓL-TÖRÖL(L, x) eltávolít egy x elemet az L láncolt listából. A törléshez meg kell adni egy x-re mutató pointert. Az eljárás ekkor a mutatók megfelelő állításával „kifűzi" x-et a listából. Ha egy adott kulcsú elemet akarunk törölni, akkor előbb végre kell hajtani a LISTÁBAN-KERES eljárást, amely visszaadja a törlendő elem pointerét. A művelethez ekkor a legrosszabb esetben $\Theta(\mathrm{n})$ idő szükséges.

A korábbi három eljárás kódja egyszerűsödne, ha nem kellene figyelni a lista két végére. A listát ezért gyakran kiegészítjük olyan speciális rekordokkal, amelyek szerepe, hogy egyszerűsítse a lista határainak ellenőrzését. Tegyük fel például, hogy az L lista tartalmaz egy $nil[L]$ mutatóval azonosított elemet. A műveletek kódjában ezután a NIL minden előfordulását a $nil[L]$-lel helyettesítjük. Ekkor például egy kétirányú lista olyan ciklikus listává alakul át, amelyben a speciális $nil[L]$ elem a lista első és utolsó eleme között található, tehát a $köv[nil[L]]$ mutató a lista fejére, az $előző[nil[L]]$ pointer pedig az utolsó elemre mutat. Ezzel összhangban az utolsó elem $köv$ és az első elem $előző$ mezője egyaránt a

$nil[L]$-re mutat. Ekkor azonban a $fej[L]$-re nincs szükségünk, mivel minden rá vonatkozó hivatkozást tudunk helyettesíteni. Az üres lista csak a speciális elemet tartalmazza, és annak mindkét pointere ugyanerre az elemre mutat, azaz $köv[nil[L]] = nil[L]$ és $előző[nil[L]] = nil[L]$. A speciális rekordok használata ritkán csökkenti az adatszerkezetek műveleteire adható aszimptotikus időkorlátot, de az azokban szereplő állandókat azonban igen. Nem is annyira a futási idő javítása miatt használjuk őket, hanem mert általuk áttekinthetőbb és érthetőbb lesz a műveletek kódja. A láncolt lista esetén a műveltek kódja egyszerűsödik, de nem sikerült jelentős időt megtakarítani. Más esetekben azonban a használatukkal lényegesen rövidíthető a kód, ezáltal csökkenthető a futási idő. Vannak olyan esetek, amikor nem célszerű speciális rekordokat alkalmazni. Például ha sok kisméretű listával dolgozunk, akkor ezek együttesen már jelentős memóriát foglalhatnak el.

2. Felhasználói dokumentáció

2.1. Adott csúcsból induló legrövidebb utak problémája

Például: Adott egy térképünk, amelyen fel vannak tüntetve az egyes kereszteződések távolságai. Ekkor hogyan juthat el egy sofőr az egyik városból a másikba a lehető legrövidebb úton?

A fenti példa probléma egy speciális esete, amelynek egyik megoldása, hogy megvizsgáljuk az összes utat az adott két város között, és kiválasztjuk a legkisebb távolságút. Csakhogy általában két város között több millió út is szerepelhet, amelyek közül sok nem is érdemel figyelmet, hiszen óriási kerülőkkel tennék meg a távolságot. Az utak száma gráf méretében exponenciális méretű is lehet.

Ezt a feladatot megfogalmazhatjuk gráfok segítségével is, azaz a legrövidebb utak problémában adott egy élsúlyozott, irányított $G = (V, E)$ gráf, ahol $\omega : E \rightarrow \mathcal{R}$ súlyfüggvény rendel az élekhez valós értékeket. A $p =< v_0, v_1, \dots, v_k >$ út súlya az utat alkotó élek súlyainak az összege: $\omega(p) = \sum_{i=1}^{k} \omega(v_{i-1}, v_i)$.

Definiáljuk az u-ból v-be vezető legrövidebb út súlyát:

$$\delta(u, v) = \begin{cases} \min\left\{w(p) : u \overset{p}{\rightarrow} v\right\}, ha\ vezet\ út\ u - ból\ v - be, \\ \qquad \infty\ , különben \end{cases}$$

Egy az u csúcsból a v csúcsba vezető legrövidebb úton a $\delta(u, v)$ súlyú u-ból v-be vezető utakat értjük. A korábbi példára lefordítva, a térképet egy gráffal modelleztük, amelyben az élek a kereszteződések között fekvő utak, az élsúlyok az útszakaszok hosszát, a csúcsok pedig a kereszteződéseket jelentik. Előfordulhat, hogy az élsúlyok nem mindig távolságot fejeznek ki, hanem használják idő, költség, veszteség megjelenítésére is.

A fenti példánál, a sofőr hiába ismeri a legrövidebb út hosszát, ettől még nem fogja ismerni az utat. Ezért gyakran szükségünk van az út menti „kereszteződésekre”, azaz a csúcsokra is, hiszen így már pontosan meg tudjuk határozni magát az utat is. Egy adott $G = (V, E)$ gráf minden $v \in V$ csúcsához hozzárendelünk egy $\pi[v]$ szülő értéket, amely vagy a csúcs egyik szülőjére, vagy a NIL-re mutat. A Dijkstra legrövidebb-utak algoritmusnál a π értékek mentén egy v csúcsból visszafelé haladva megkapjuk az egyik s-ből v-be vezető legrövidebb utat. Egy ilyen algoritmus működése során a π értékek nem feltétlenül adják meg

a legrövidebb utakat, ezért bennünket igazán a $G_\pi = (V_\pi, E_\pi)$ szülő részgráf érdekel, amelyet a π értékek határoznak meg. A V_π nem NIL szülőjű G-beli csúcsokat és az s (kezdőcsúcs) csúcsot tartalmazza: $V_\pi = \{v \in V : \pi[v] \neq \mathrm{NIL}\} \cup \{s\}$. Az E_π-beli irányított élek a π értékek által kijelölt csúcsokból vezetnek a V_π-beli csúcsokba: $E_\pi = \{(\pi[v], v) \in E : v \neq V_\pi\} - \{s\}\}$. Belátható hogy a Dijkstra-algoritmus olyan π értékeket állít elő, hogy azok befejezésekor a G_π egy „legrövidebb-utak fája" lesz – egy olyan s gyökerű fa, amely az s-ből elérhető bármely G-beli csúcsba egy s-ből kiinduló legrövidebb utat tartalmaz. Legyen $G = (V, E)$ egy irányított, $\omega : E \rightarrow \boldsymbol{R}$ súlyfüggvénnyel élsúlyozott gráf, és tegyük fel, hogy a G nem tartalmaz az s kezdőcsúcsból elérhető negatív köröket. Ekkor a legrövidebb utak jól definiáltak. Egy s gyökerű legrövidebb-utak fa egy olyan $G' = (V', E')$ részgráf, ahol $V' \subseteq V$ és $E' \subseteq E$ úgy, hogy V' az s-ből elérhető G-beli csúcsok halmaza, G' egy s gyökerű fa, és minden $v \in V'$-re az egyetlen G'-beli s-ből vezető út egy legrövidebb s-ből v-be vezető út G-ben. A legrövidebb utak nem szükségszerűen egyértelműek, csak úgy, mint a legrövidebb-utak fái.

A Dijkstra-algoritmus a fokozatos közelítés módszerén alapul, amely lényege, hogy lépésről-lépésre csökkenti az egyes csúcsok legrövidebb-út súlyának felső korlátját, amíg ez megegyezik a csúcs legrövidebb-út súlyával. Ezért minden $v \in V$ csúcsnál eltárolunk egy $d[v]$ értéket, amely az s kezdőcsúcsból a v-be vezető legrövidebb út súlyának felső korlátját tartalmazza. A $d[v]$-t egy legrövidebb-út becslésnek nevezzük. A legrövidebb-út becsléseket és a szülőkre mutató π értékeket a következő eljárás állítja be.

EGY-FORRÁS-KEZDŐÉRTÉK(G, s)

for minden $v \in V[G]$-re
  do
    $d[v] \leftarrow \infty$
    $\pi[v] \leftarrow NIL$
$d[s] \leftarrow 0$

Amikor egy (u,v) éllel próbálunk közelíteni, akkor meg kell vizsgálni a v csúcshoz az u-n keresztül eddig talált legrövidebb utat, és ha a vizsgált út rövidebb, mint az eddig nyilvántartott, akkor módosítja a $d[v]$ és a $\pi[v]$ értékeket. Ez a közelítő lépés csökkenti a $d[v]$ legrövidebb-út becslés értékét és átállítja a $\pi[v]$ mezőt az u csúcsra.

KÖZELÍT(u, v, w)

if $d[v] > d[u] + w(u, v)$

then

  $d[v] \leftarrow d[u] + w(u, v)$

  $\pi[v] \leftarrow u$

Amikor egy közelítő lépéssorozat kiszámolja a tényleges legrövidebb-út súlyokat, akkor a π értékek eredményeként előállt G_π szülő részgráf egy legrövidebb-utak fája lesz a G-ben.

2.1.1. DIJKSTRA-ALGORITMUS

Mint már korábban említettem a Dijkstra-algoritmus is az adott kezdőcsúcsból induló legrövidebb utak problémáját megoldó algoritmus. Csak abban az esetben oldja meg a problémát, ha a vizsgált élsúlyozott, irányított $G = (V, E)$ gráfban egyik élnek sem negatív a súlya. Az algoritmus azoknak a csúcsoknak az S halmazát tartja nyílván, amelyekhez már meghatározta az s kezdőcsúcsból odavezető legrövidebb-út súlyát. Azaz minden $v \in S$ csúcsra a $d[v] = \delta(s, v)$. Ezután már miden lépésben kiválasztja a legkisebb legrövidebb-út becslésű $u \in V - S$ csúcsot, beteszi az S-be, és miden u-ból kivezető éllel egy-egy közelítést végez. Az algoritmust egy elsőbbségi sor konkrét megvalósításán keresztül mutatom be, amely $V - S$-beli csúcsok tárolására szolgál, és azok d értékeivel indexeli őket. Természetesen más típusú adatstruktúrát is lehet alkalmazni, ilyenekre mutat példát a szakdolgozat későbbi része. Továbbá feltételezem azt is, hogy a G gráf egy szomszédsági listával van megadva.

DIJKSTRA(G, s)

EGY-FORRÁS-KEZDŐÉRTÉK(G, s)

$S \leftarrow \emptyset$

$Q \leftarrow V[G]$

while $Q \neq \emptyset$

do

  $u \leftarrow$ KIVESZ-MIN(Q)

  $S \leftarrow S \cup \{u\}$

  for minden $v \in Szomszéd[u]$-ra

    do KÖZELIT(u, v, w)

Az 1. sorban a d és π értékek korábban leírt kezdeti inicializálását végzi el az algoritmus. A 2. sorban üressé teszi az S halmazt, majd a 3.-ban pedig a Q prioritásos sort készíti elő úgy, hogy minden $V - S = V - \emptyset = V$-beli csúcsot tartalmazzon. A while ciklusban kivesz mindig egy u csúcsot a $Q = V - S$-ből, és berakja az S-be. A for ciklusban pedig a közelítést végzi az u-ból kivezető (u, v) élekkel, ezáltal módosítják a $d[v]$ becslést és a $\pi[v]$ szülőt, feltéve, hogy a v-hez az u-n keresztül most talált út rövidebb, mint az eddig nyilvántartott. Jobban megvizsgálva az algoritmust, látható, hogy a 3. sor után már nem kerül be csúcs Q-ba, illetve, hogy mindegyik csúcsot kivesszük Q-ból és aztán áttesszük S-be, tehát ennek megfelelően a while ciklus pontosan $|V|$-szer hajtódik végre. A Dijkstra-algoritmus mohó stratégiát alkalmaz, hiszen mindig a „legkönnyebb", a „legközelebbi" csúcsot választja ki $V - S$-ből, hogy azután betegye az S halmazba. Sajnos a mohó stratégiák nem mindig adnak optimális eredményt, de az algoritmus mindig a legrövidebb utakat állítja elő. Azaz belátható a Dijkstra-algoritmus helyessége is.

Miután megterveztünk egy algoritmust, mindig a következő lépés annak tesztelése, hogy milyen gyors, hol lehetne még javítani, optimalizálni rajta. Most is a fő kérdés az algoritmus műveletigénye?

Vegyük először azt az esetet, amelyben Q prioritásos sort egy lineáris tömbbel valósítjuk meg. Ekkor miden KIVESZ-MIN művelet $O(V)$ ideig tart és pontosan $|V|$ ilyen művelet van, ezért a KIVESZ-MIN teljes műveletigénye $O(V^2)$. Mindegyik $v \in V$ csúcsot pontosan egyszer tesszük be az S halmazba, így a $szomszéd[v]$ szomszédsági lista mindegyik élét egyszer vizsgálja meg a for ciklus az algoritmus működése folyamán. Mivel szomszédsági lista összes élének száma $|E|$, a $O(1)$ ideig tartó for ciklus iteráció is ennyiszer fut le. Így a futási ideje az egész algoritmusnak $O(V^2 + E) = O(V^2)$. Egy másik eset, amikor a gráf ritka, akkor a Q prioritásos sort érdemesebb bináris kupaccal megvalósítani. Az így kapott algoritmust néha módosított Dijkstra-algoritmusnak is nevezik. Mindegyik KIVESZ-MIN művelet futtatási ideje $O(\lg(V))$. Most is $|V|$ ilyen művelet van. A bináris kupac felépítésének ideje $O(V)$. A KÖZELIT $d[v] \leftarrow d[u] + w(u, v)$ értékadását a KULCSOT-CSÖKKENT$\big(Q, v, d[u] + w(u, v)\big)$ hívása végzi, amely $O(\lg(V))$ ideig tart, és legfeljebb $|E|$ ilyen művelet van. A teljes futási idő így $O\big((V + E) * \lg(V)\big)$, ami $O(E * \lg(V))$, ha mindegyik csúcs elérhető a kezdőcsúcsból. Valójában a $O(V * \lg(V) + E)$ futási időt is elérhetjük, ha Q prioritásos sort Fibonacci-kupaccal implementáljuk. Az $|V|$ KIVESZ-MIN művelet amortizációs ideje $O(\lg(V))$, és az $|E|$ KULCSOT-CSÖKKENT művelet mindegyike $O(1)$ amortizációs idejű. A Fibonacci kupac felfedezését annak a módosított Dijkstra-algoritmusnak a tanulmányozásának köszönhetjük, ahol potenciálisan sokkal több KULCSOT-CSÖKKENT művelet hajtódik végre, mint KIVESZ-MIN, így minden olyan módszer, amely minden KULCSOT-CSÖKKENT műveletet $o(\lg(V))$

amortizációs idejűre redukál miközben a KIVESZ-MIN amortizációs idejét nem növeli, aszimptotikusan gyorsabb implementációt eredményez.

2.2. A LEMON könyvtár

A kupacok a LEMON könyvtár segítségével készültek, amely gráf elméleti problémák megoldására készített C++ könyvtár. Célja, hogy a gráfelméleti algoritmusokat egyszerre hatékonyan és egyszerűen lehessen használni. A LEMON használatához linux operációs rendszerszükséges. A következő linux kódsorok segítségével letölthető, illetve konfigurálható:

```
svn co https://lemon.cs.elte.hu/svn/lemon/trunk lemon     # letölti egy lemon nevű könyvtárba
cd lemon
autoreconf -vi
./configure                 # konfigurálás
make                        # fordítás
[ make check                # ellenőrzés ]
make doc                    # dokumentáció generálása
```

A dokumentációt a Doxygen program végzi, így a forráskódban a dokumentációs megjegyzések készítéséhez e program konvencióit kell követni.

A rendszer frissíthető az svn up lemon paranccsal. Természetesen ezek után újrafordítás szükséges a make utasítással.

Ezek után térjünk rá a Dijkstra-algoritmus futtatására.

```
SmartGraph graph;

SmartGraph::EdgeMap<int> cost(graph);
SmartGraph::Node source;

GraphReader<SmartGraph>(argv[1], graph).
  readEdgeMap("cost", cost).
  readNode("source", source).
  run();
{
  Dijkstra<SmartGraph, SmartGraph::EdgeMap<int> >::
    DefStandardHeap<FouraryHeap<int, SmartGraph::NodeMap<int> > >::
    Create dijkstra(graph, cost);
  Timer timer;
  dijkstra.run(source);
  cout << "4-ary heap: " << timer << endl;
}
```

Először is létre kell hozni egy gráfot, majd fel kell „tölteni” adatokkal. Jelen esetben egy fájlból kapja az értékeket, melyet paraméterül kap meg. A gráfot tartalmazó fájlnak a LEMON rendszerben használt lgf kiterjesztésű fájlok konvencióit kell követnie. Majd át kell adni a Dijkstra-nak paraméterként a gráfot, az élek listáját,

végül pedig egy kupacot, melyet ott definiálunk is. A kupac definiálásához meg kell adni a Prio értékek típusát (int), és a csúcsok Map-jét (NodeMap). A create-tel létrehozzuk a Dijkstra-algoritmust, majd run-nal futtatjuk a source kezdőcsúcsra. Ezen kívül egy timer segítségével az algoritmus futási idejét is lehet mérni, hiszen, mint már említettem igen fontos a gyors futási idő.

Példa a fenti tesztfájl lefordítására:

```
g++ -O2 lemon/kacsi/heap_test_dijkstra.cpp -o lemon/kacsi/heap_test_dijkstra
    -I lemon -L lemon/lemon/.libs/ -lemon
```

A fordításnál a -O2 flag az optimalizálásra utal, aztán megadom a fordítandó fájl elérési útvonalát, majd –o flag-gel az kimenetet, aztán include-olom a lemon-t. Meg kell adni még további fájlokat, melyek szükségesek a fordításhoz. A –L-lel az elérési útvonalat, majd a –l után pedig a nevüket kell megadni. Ekkor a fájlok neveiből elhagyható a lib előtag. Érdekességként, azért nevezték el őket emon-nak, hogy a –l flag-gel együtt lemon-t alkosson.

3. Fejlesztői dokumentáció

3.1. Kupac-implementációk

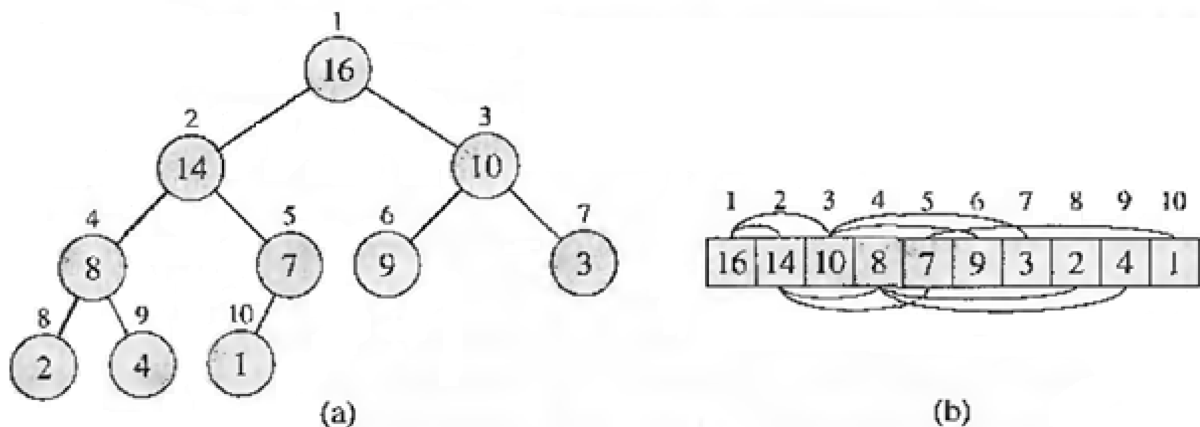
Továbbiakban a már létező és az általam megvalósított kupacokról lesz szó, ezek leírásáról, összehasonlításáról. A kupacok a LEMON könyvtár segítségével készültek, amely gráf elméleti problémák megoldására készített C++ könyvtár. Célja, hogy a gráfelméleti algoritmusokat egyszerre hatékonyan és egyszerűen lehessen használni, ezért a könyvtár generikus (template) programozási módszertan szerint készült. Tehát az összes kupac sablon, amelyeket kívülről kell a megfelelő adatokkal meghívni. A sablon előnye hogy gyorsan lehet meghívni. Ha objektum orientált programozási módszerrel, azaz öröklődéssel, virtuális függvényekkel implementáltam volna a kupacokat, akkor azok nagyon lassúak lennének, és sokszori meghívásuk jelentősen rontaná a futási időt. Mivel a futási idő az egyik legfontosabb szempont, ezért is választották a LEMON alkotói a generikus programozási módszertant.

Minden LEMON-ban megvalósított kupacnak meg kell felelnie bizonyos formai követelményeknek. Ezek a formai követelmények ellenőrzésére készült a concepts header fájl. Minden kupac esetében először paraméterként kell megadni a kulcs érték (Prio), és a gráf elemeit tartalmazó Map típusát. Ezen kívül megadhatjuk az összehasonlítás feltételét, de alapértelmezett értéke kisebb. A megadott típusokra a typedef kulcsszóval azonosítókat definiálunk. Tartalmaznia kell egy felsorolási típust (state), mely egy adott csúcs állapotára utal. Az állapota lehet: még nincs a kupacban (-1), benne van (0), már nincs benne (-2). Természetesen kell egy konstruktor, ami egy olyan LEMON map-et vár referenciaként paraméternek, ami minden kulcshoz egy egész értéket rendel, és ezt a map-et a kupac belső adattárolásra használhatja fel. A konstruktort érdemes explicit-nek deklarálni, így csökkenthetjük a rossz paraméterezésből származó hibákat. Szükség van néhány további kívülről elérhető függvényre is. Ilyen a SIZE, amely a csúcsok számát, a TOP, amely a minimum csúcs nevét, és a PRIO, amely pedig az értékét adja vissza. Ezen kívül kell egy EMPTY logikai típussal visszatérő művelet, amely megmondja, hogy a kupacunk üres-e. Található még egy SET eljárás, amely először megvizsgálja a kapott Item adatot, hogy bent van-e már a kupacban, ha benne van, akkor az eredeti és az új kulcs értéktől függően változtatja az elem kulcsát, és rendezi a kupacot. Továbbá nem elhagyhatók a Dijkstra-algoritmus szempontjából legfontosabb publikus műveletek a PUSH (beszúr), POP (minimumot-kivág), DECREASE (kulcsot-csökkent), INCREASE (kulcsot-növel). Ezeket az adattagokat, eljárásokat és függvényeket kell tartalmaznia minden kupac implementációnak.

Minden elkészített header fájlt be kell tenni a makefile.am fáljba, hogy a make parancs kiadásakor már azt a fájlt is vegye figyelembe a LEMON csomag elkészítésénél.

3.1.1. BINÁRIS, 4-GYEREKES, K-GYEREKES KUPAC

A bináris kupac adatszerkezet úgyis elképzelhető, mint egy majdnem teljes bináris fa egy tömbben ábrázolva. A fa minden csúcsa a tömb egy eleme, mely a csúcs értékét tárolja. A fa minden szintjén teljesen kitöltött, kivéve a legalacsonyabb szintet, ahol balról jobbra haladva csak egy adott csúcsig vannak elemek. Egy A tömb, mely egy kupacot alkot, két tulajdonsággal rendelkezik: *hossz*[A], mely a tömb elemeinek száma, és *kupac-méret[A]*, mely az A tömbben tárolt kupac elemeinek száma. Így A[1…*hossz[A]*] minden eleme tartalmazhat érvényes számot, A[*kupac-méret[A]*] utáni értékek, ahol *kupac-méret[A]* ≤ *hossz[A]*, már nem tartoznak a kupachoz. A fa gyökere A[1], és ha i a fa egy adott csúcsának tömbbeli indexe, akkor az ősének *szülő(i)* = $\lfloor i/2 \rfloor$ baloldali gyerekének *bal(i)* = $2 * i$, és jobb oldali gyerekének *jobb(i)* = $2 * i + 1$ az indexe.



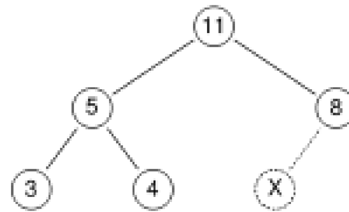
A bináris kupacnak két fajtája van: a maximum-kupac és a minimum-kupac. Mindkét fajta kupacban a csúcsok értékei kielégítik a kupactulajdonságot, melynek jellemzői függnek a kupac fajtájától. Maximum-kupac esetén a kupac minden gyökértől különböző i eleme maximum-kupactulajdonságú: A[*szülő(i)*] ≥ A[i], azaz az elem értéke legfeljebb akkora, mint a szülőjének értéke. Így a maximum-kupac legnagyobb eleme a gyökér, és egy adott csúcs alatti részfa minden elemének értéke nem nagyobb, mint az adott csúcsban levő elem értéke. A minimum-kupac épp ellentétes szerkezetű, tehát a kupac minden gyökértől különböző i eleme minimum-kupactulajdonságú: A[*szülő(i)*] ≤ A[i]. A minimum-kupacban a legkisebb elem van a gyökérben. A kupacrendezés algoritmusban maximum-kupacot, az elsőbbségi soroknál pedig általában a minimum-kupacot alkalmazzák.

Ezt a kupacot már megvalósították. Ennek ellenére azért írtam le mégis a jellemzőit, mivel a 4-gyerekes és a K-gyerekes kupac is ezen alapul. A bináris mintájára keletkeleztek, csak itt egy csúcsnak 4 illetve K gyereke van, és nem 2. Tehát implementáltam a 4-gyerekes, és az általános K-gyerekes kupacot is. A továbbiakban ezek leírását és összehasonlítását tárgyalom.

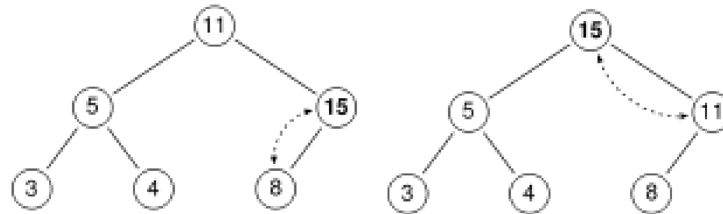
Az elemeket egy data nevű vektorban tárolom, amely Pair típusú elemeket tartalmaz. A Pair egy általam definiált típus, mely Item és Prio elemekből állnak. Az Item az elem neve, a Prio pedig az elem értékét tárolja. E két kupac esetében is hasonlóan lehet meghatározni a gyereket és a szülőt, mint a bináris kupac esetében. Például K gyerek esetén a szülőt $(i-1)/K$ (a C++-ban a / művelet rögtön alsó egészrészt is jelent), az első gyereket pedig $K*i+1$ képlettel számolhatjuk ki. A némi eltérés oka, hogy az implementációban az adatokat tartalmazó data vektor indexelése nem 1-től, hanem 0-tól indul.

Akkor térjünk rá a beszúrás és a minimum-törlés műveletekre. Egy elem beszúrása a PUSH nevű eljárással történik, melynek a bemenő paramétere egy Pair típusú elem. Ekkor először növeli a data méretét 1-gyel, majd meghívja a BUBBLE_UP eljárást, amely az elemet a megfelelő helyre buborékoltatja fel. Műveletigénye megegyezik a bináris kupacéval mindkettő esetében, azaz $\Theta(lg\ n)$. A minimum elem eldobását, a POP végzi, melynek nincs bemenő paramétere, hiszen a minimum helyét ismeri. Először csökkenti a data méretét 1-gyel, majd az utolsó elemet beteszi a volt minimum helyére, ezután a BUBBLE_DOWN procedúra újra helyreállítja a kupactulajdonságot. Műveletigénye ugyanakkor már jobb mint a bináris halomnál, mivel 4-ary esetében a fa magassága fele akkora. Ezért a BUBBLE_DOWN eljárás fele annyi ideig fut, így a POP is, így lesz $\Theta((lgn)/2)$, hasonlóan a K-gyerekes esetén $\Theta((lgn)/K)$. Belátható, hogy az elméleti legjobb választás a K-ra 2+m/n. A POP-nál nincs szükség a minimum elem megkeresésére, mivel az $\Theta(1)$ időben megtalálható, hiszen mindig a data vektor első eleme lesz, mivel az a gyökérelem.

Egy elem beszúrása (push)

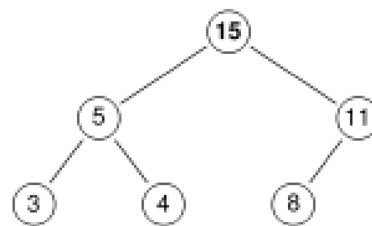


Az X helyére 15 szúrunk be

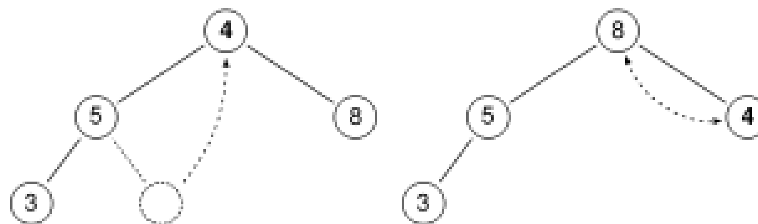


A beszúrt elemet felfelé buborékoltatjuk

Minumum eltávolítása (pop)



Eltavolítjuk a gyökérelemet



Az utolsó elemet felvisszük gyökérnek, majd lefelé buborékoltatjuk

További két fő művelet a Dijkstra-algoritmus szempontjából az INCREASE (kulcsot-növel), és a DECREASE (kulcsot-csökkent) műveletek. A DECREASE procedúra egy adott elem Prio értékét csökkenti meg, majd buborékoltatja fel a helyére. Az elemet és az új kulcsértéket paraméterül kapja meg. A műveletigénye szintén megegyezik a bináriséval, azaz $\Theta(lg\, n)$. Az INCREASE eljárás pedig az előző ellentettje, azaz növeli az adott elem Prio értékét, végül azt lefelé buborékoltatja. Továbbá le lehet kérdezni a minimum Item és Prio értékeit, erre szolgál a PRIO és az TOP függvények, de ezeket a bevezetőben már említettem.

Ezek voltak a Dijkstra-algoritmus szempontjából főbb publikus eljárások, függvények.

Ezek után tárgyalom a főbb privát műveleteket.

```
void bubble_up(int hole, Pair p) {
  int par = parent(hole);
  while( hole>0 && less(p,data[par]) ) {
    move(data[par],hole);
    hole = par;
    par = parent(hole);
  }
  move(p, hole);
}
```

A BUBBLE_UP eljárásnak két bemenő paramétere van. Az első, a hole int típusú, amely az elem data vektorban elfoglalt helye, a p pedig Pair típusú tag, mely tartalmazza az elem Item, és az új Prio (kulcs) tagjait. Az új kulcs (prio) érték mindig kisebb, mint eredeti érték. Emiatt az elemet addig kell felfelé vinni, amíg nem lesz kisebb a szülő kulcs értékénél. Erre a kupactulajdonság megtartása miatt van szükség. Amennyiben nagyobb a szülő értéke, akkor a szülőt elhelyezi a kapott elem eredeti helyén (hole), majd a gyerek megkapja a szülő helyét és így tovább. Tehát először meghatározza az eredeti szülő eredeti helyét és eltárolja a par nevű változóban. Majd egy while ciklussal addig végzi az elemcseréket, amíg a hole változó nagyobb 0, azaz nem kerültünk a gyökérbe, illetve amíg nagyobb a szülő kulcs értéke. Az utóbbit vizsgálja a LESS függvény. A csere folyamata tulajdonképpen a szülőt a hole értékének megfelelő helyre teszi a vektorban a MOVE segédeljárás segítségével, majd a hole megkapja a szülő volt helyét és meghatározza az új szülőt, amit eltárolunk a par változóban. Végül, ha már megtalálta a kapott elem jó helyét, akkor ismét a MOVE odarakja az elemet. „Jó” hely alatt olyan helyet értek, amelyre ha az elemet helyezi a kupactulajdonság teljesülni fog.

```
void bubble_down(int hole, Pair p, int length) {
  if( length>1 ) {
    int child = first_child(hole);
    while( child<length ) {
      child = find_min(child, length);
      if( !less(data[child], p) )
        goto ok;
      move(data[child], hole);
      hole = child;
      child = first_child(hole);
    }
  }
  ok:
    move(p, hole);
}
```

A BUBBLE_DOWN eljárás az előbb említett ellentettje. Itt viszont három bemenő paraméterre van szükség, kiegészülve a data vektor hosszával (length). Erre azért

nem volt szükség az előző esetén, mivel ott ismerte a vektor elejét, hiszen a 0. helyen található az első elem. A p elem kulcs értéke itt nagyobb lesz, mint az eredeti. Most azonban nem a szülő értékét kell vizsgálni, hanem a gyerekekét. Tehát első lépésként megvizsgálja, hogy a vektor hossza, azaz a kupacban levő elemek száma nagyobb-e, mint 1. Hiszen ha nem nagyobb, akkor felesleges a ciklus, mert csak egy helyre helyezhető az elem. Ha teljesül a feltétel, akkor meghatározza a hole helyen levő elem első gyerekét, és eltárolja a child nevű változóban. A ciklus addig fut, míg el nem éri az adatok végét. A ciklusmag határozza meg a „jó” helyet, ahova a helyezheti az elemet. Először meg kell határozni a gyerekek közül a legkisebbet, mivel ha az kerül gyökérbe, akkor megmarad a rendezettség. Ezt végzi a FIND_MIN függvény. A visszaadott értéket is el kell tárolni. Amennyiben kisebb a gyerek kulcs értéke, mint a kapotté, akkor a hole változóban tárolt helyre helyezi a gyermeket a vektorban, és a hole megkapja a gyerek helyét. Ezután meghatározza az új első gyereket. Erre azért van szükség, mert a FIND_MIN függvénynek paraméterül kell adni. Amikor a program kilép a ciklusból, akkor már csak a megfelelő helyre kell helyezni a kapott elemet. Eddig a műveletek a két általam megvalósított és a bináris kupacnál is megegyeztek.

A következő FIND_MIN függvény megvalósítása azonban már eltér, hiszen a binárisnál kettő, a 4-gyerekes esetén négy, a K-gyerekes esetén pedig K gyerekből kell kiválasztani a minimumot. K gyerek esetén a következőképpen néz ki a művelet:

```
int find_min(const int child, const int length) {
  int min=child, i=1;
  while( i<K && child+i<length ) {
    if( less(data[child+i], data[min]) )
      min=child+i;
    ++i;
  }
  return min;
}
```

A bemenő paraméterei egy gyerek (const int child), ami az első gyerek lesz és egy hossz (const int length). A paraméterek nyugodtan lehetnek konstansok, hiszen nem változtatja meg értéküket. A min lokális változó megkapja rögtön a kapott első gyerek értékét, azaz azt tekinti minimumnak. Majd egy while ciklussal végig megy a gyereken, miközben figyeli az adatokat tartalmazó vektor végét. A ciklusban vizsgálja, hogy a gyerek kulcs (Prio) értéke kisebb-e, mint a min változó értéke helyén szereplő elem kulcs értéke. Ha igen, akkor ő lesz az új minimum. Majd a ciklus végeztével visszatér a min értékével. Ugyanezt a műveletet a 4-gyerekes halom esetében már nem ciklussal valósítottuk meg, mivel az túlságosan lassú megoldás lett volna.

```
int find_min(const int child, const int length) {
  int min=child;
  if( child+3<length ) {
    if( less(data[child+3], data[min]) )
      min=child+3;
    if( less(data[child+2], data[min]) )
      min=child+2;
    if( less(data[child+1], data[min]) )
      min=child+1;
  }
  else {
    if( child+2<length ) {
      if( less(data[child+2], data[min]) )
        min=child+2;
      if( less(data[child+1], data[min]) )
        min=child+1;
    }
    else {
      if( child+1<length ) {
        if( less(data[child+1], data[min]) )
          min=child+1;
      }
    }
  }
  return min;
}
```

Ezért inkább több elágazással oldottuk meg. A bemenő paraméterek megegyeznek az előzővel. Szintén használ egy min nevű lokális változót a minimum tárolására. Először vizsgálja a gyerekek számát, ha a harmadik gyerek benne van a data vektorban, akkor végig kell menni mindhárom gyermeken, és vizsgálni kell melyik a legkisebb. Amennyiben nincs három gyermek, akkor vizsgájuk kettőre, és így tovább. Végül pedig visszatérünk a legkisebb gyerekkel (min).

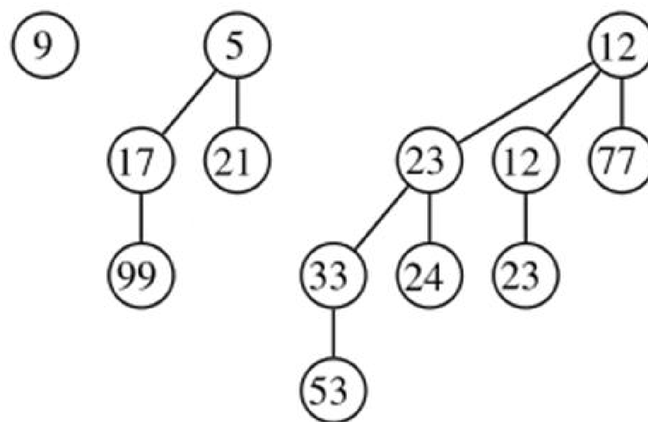
A kupacok viselkedhetnek maximum- és minimum-kupactulajdonságú halomként is. Ugyanakkor az elnevezések nem erre utalnak (pl.: FIND_MIN). Ennek ellenére azért neveztem el így a műveleteket, mivel a Dijkstra-algoritmus minimum-tulajdonságú kupacot használ, és főként ezen algoritmus használatához készültek.

3.1.2. BINOMIÁLIS KUPAC

A binomiális fa egy rekurzív módon definiált rendezett fa. A B_0 binomiális fa egy csúcsból áll. A B_k binomiális fa két összekapcsolt B_{k-1} binomiális fából áll. Az egyik fa gyökércsúcsa a másik fa gyökércsúcsának legbaloldalibb gyereke. Egy H binomiális kupac binomiális fák olyan halmaza, amely kielégíti a következő binomiális-kupac tulajdonságokat.

1. H minden binomiális fája rendelkezik a min-kupac tulajdonsággal: egy csúcs kulcsa nagyobb vagy egyenlő, mint a szülőjének a kulcsa. Ekkor azt mondjuk, hogy ezek a fák min-kupac-rendezett fák.
2. H-ban legfeljebb egy olyan binomiális fa van, amelyikben a gyökércsúcsnak egy meghatározott fokszáma van.

Az első tulajdonság azt mondja ki, hogy egy min-kupac-rendezett fa gyökércsúcsában van a fa legkisebb kulcsa. A második tulajdonságból az következik, hogy egy n csúcsból álló H binomiális kupac legfeljebb $\lfloor \lg n + 1 \rfloor$ binomiális fából áll.



Példa 13 elemet tartalmazó binomiális kupacra, amely tartalmaz 3 binomiális fát, melyek foka sorrendben 0,2 és 3.

Egy binomiális kupacban levő binomiális fákat baloldali gyerek, jobboldali testvér-ábrázolással adjuk meg. Mindegyik csúcsnak van egy kulcs mezője. Ezen kívül minden x csúcsban van egy *p[x]* mutató, amelyik a csúcs szülőjére, egy *gyerek[x]* mutató, amelyik a legbaloldalibb gyerekére, és egy *testvér[x]* mutató, amelyik az x jobb oldalán álló első testvérére mutat. Ha az x csúcsnak nincs gyereke, akkor a *gyerek[x]* = NIL, és ha az x a szülőjének a legjobboldalibb gyereke, akkor a is *testvér[x]* = NIL. Mindegyik x csúcsnak van *fokszám[x]* mezője is, amelyik az x gyerekeinek számát tartalmazza. Egy binomiális kupac binomiális fáinak a gyökércsúcsai egy láncolt listába vannak fűzve, ezt a listát gyökérlistának nevezzük. A gyökérlista bejárásakor a gyökércsúcsok fokszámai szigorúan növekednek. A binomiális kupacok második tulajdonsága alapján, ha egy binomiális kupacnak n csúcsa van, akkor a gyökércsúcsok fokszámainak halmaza a {0, 1, . . . , $\lfloor \lg n \rfloor$} halmaz részhalmaza. A gyökércsúcsokban a testvér mezőnek más a jelentése, mint a nem gyökér csúcsokban. Ha x egy gyökércsúcs, akkor a *testvér[x]* a gyökérlista következő elemére mutat. Ha x a gyökérlista utolsó eleme, akkor természetesen a *testvér*[x] = NIL. Egy adott H binomiális kupac *fejelem[H]* mezővel címezhető meg, ez a H gyökérlistájának első gyökércsúcsára mutató

pointer. Ha egy H binomiális kupacnak nincs egy eleme sem, akkor *fejelem[H]* = NIL.

Az implementációban az előzőleg megadott elvet követtem, csak a pointereket nem pointerekkel, hanem int típussal azonosítottam. Amelyik elemre mutat a pointer, a neki megfelelő változó azon elem a container nevű vektorban elfoglalt helyét tartalmazza. Így például a child pointer által mutatott elemre container[child] néven hivatkozhatunk.

```
class store {
  friend class BinomHeap;

  Item name;
  int parent;
  int right_neighbor;
  int child;
  int degree;
  bool in;
  Prio prio;

  store() : parent(-1), right_neighbor(-1), child(-1), degree(0), in(true) {}
};
```

Egy store nevű osztályban reprezentálom a csúcsokat. Tartalmaz egy friend osztályt, amire azért volt szükség, hogy a BinomHeap osztály is elérje az adattagokat. A korábban már említett kulcs mezőt a prio, a fokszámot a degree, a szülő, a gyerek és a testvér pointert pedig sorrendben a parent, a child és a right_neighbor int típusú változókkal reprezentálom. Az implementációban a NIL-t (-1)-gyel azonosítom. Továbbá minden csúcsnak van egy Item, amely a neve lesz, és egy bool típusú adattagja, amely arra utal, hogy az adott elem benne van-e a kupacban. A csúcsokat egy container nevű vektorban ábrázolom, amely elemei store osztály egyedei.

Egy elem beszúrása a PUSH eljárással, a minimális csúcs eltávolítása a POP művelettel történik. A PUSH bemenő paraméterei egy Item és Prio típusú adatok. Először létrehoz egy egyelemű kupacot, majd meghívja a MERGE procedúrát, amely a két binomiális kupacot egyesíti. Mindenekelőtt az Item típusú változó alapján azonosítja a beszúrandó elemet, azaz meghatározza a vektorban elfoglalt helyét. Amennyiben nem szerepel az elem benne, akkor létrehozza, majd beszúrja a vektor végére. Ha már szerepel, akkor csak alaphelyzetbe hozza az adattagjait. Azért fordulhat elő, hogy szerepel az elem a vektorban, de a fában nem, mivel amikor a fából kitöröl egy elemet, akkor a vektorból nem töröli ki, csak az in nevű bool változót állítja hamisra. A PUSH $O(\lg n)$ időben végezhető el, mivel ugyan az egyelemű kupac létrehozása $\Theta(1)$ időben elvégezhető, de a MERGE eljárás (egyesít) $O(\lg n)$ időt vesz igénybe. A POP művelet kivágja a minimális kulcsú (Prio értékkel rendelkező) elemet. Ehhez meg kell keresni a minimális elemet, majd azt kiláncolni a gyökérlistából. Ezután a gyerekeiből egy új listát kell létrehozni fordított láncolással. Végül pedig a kapott kupacot egyesíteni kell a megmaradt kupaccal. Tehát először a minimum elem in adattagját hamisra

állítja, ezzel beállítva azt, hogy az elem már nincs a kupacban. Következő lépésként létrehozza fordított láncolással a gyereklistát. Egy while ciklus segítségével haladok végig a gyerekeken, mindig tárolva az aktuális elem bal és jobb szomszédját, a prev és a neighb változókban. Az adott gyerek szülőjét NIL-re, azaz (-1)-re állítja, majd a testvérnek pedig a prev változó által mutatott elemet kapja meg. A végén pedig csúsztatja a változókat, azaz a prev lesz a child és a child lesz a neighb. Ezután ha a minimum elem az egyetlen elem a gyökérlistában, akkor az új gyökérlista a gyerek listája (head_child) lesz. Ha azonban nem egyelemű, akkor meg kell vizsgálni, hogy a minimum áll-e az első helyen, mert különben még ki is kell láncolni (unlace) az elemet a listából. Ha mindez kész, akkor a kapott két listát (maradék gyökérlistát és a gyerek listát) egyesíti (merge). A legvégén pedig megkeresi az új minimumot a FIND_MIN függvény segítségével. A POP műveletigénye $\Theta(\lg n)$.

Az INCREASE (kulcsot-növel) bemeneti paraméterei egy Item és Prio típusú érték. Az Item típus az elemre utal, a Prio pedig a kulcs értéke. Először kitörli az elemet, majd ismét beszúrja az új kulcs értékkel. Az elem törlése a következőképpen zajlik. Meghívja a KULCSOT-CSÖKKENT eljárást a minimum értékénél 1-gyel kisebb értékkel és a kitörlendő elemmel. Így az elem felkerül a gyökérlistába és ő lesz az új minimum is. Ezek után csak meg kell hívni a POP műveletet. A DECREASE (kulcsot-csökkent) eljárásnak ugyanezek a bemeneti paraméterei. Míg az előző esetben felfelé buborékoltattuk az elemet, addig itt most lefelé kell, mivel e művelet esetében a kulcs érték nagyobb lesz. Amennyiben kisebb a kapott kulcs érték a szülőjénél, amely a kapott csúcs új kulcsa lesz, akkor meg kell őket cserélni. Az eljárás először a szülő-gyerek kapcsolatokat állítja be a két csúcs új helyeinek megfelelően. Ezek után a testvér kapcsolatok jönnek. Megkeresi mindkét csúcs esetében a bal szomszédot, és a testvér pointert ráállítja az új csúcsra. Aztán a két csúcs testvér mutatóját állítja a megfelelő elemre. Amennyiben a head fejelem a szülőre mutatott, akkor át kell állítani a gyerekre. Végül folytatni kell a fenti módosításokat, amennyiben az új szülő kulcsa ismét kisebb, mint a kapott csúcsé.

A fent említett műveletek megvalósításához használtam több segéd-eljárást illetve függvényt is.

Az egyik ilyen a FIND_MIN (minimumot-keres), melynek feladata a gyökérlistán végigmenve megkeresse a legkisebb kulcsú (prio értékű) elemet. Először az első elemet eltárolja, mint minimum. Ezután csak végigmegy a lista többi elemén és összehasonlítja az aktuális minimummal.

A MERGE (egyesít) két binomiális kupacot egyesít. Mégis csak egy bemenő paramétere van, hiszen az egyik kupacra mindig a head fejelem mutat. Így a bemenő paraméter az a kupac, amellyel egyesíteni kell.

```
void merge(int a) {
  interleave(a);

  int x=head;
  if( -1!=x ) {
    int x_prev=-1, x_next=container[x].right_neighbor;
    while( -1!=x_next ) {
      if( container[x].degree!=container[x_next].degree || (
        -1!=container[x_next].right_neighbor &&
        container[container[x_next].right_neighbor].degree==container[x].degree ) ) {
            x_prev=x;
            x=x_next;
      }
      else {
        if( comp(container[x].prio,container[x_next].prio) ) {
          container[x].right_neighbor=container[x_next].right_neighbor;
          fuse(x_next,x);
        }
        else {
          if( -1==x_prev ) { head=x_next; }
          else {
            container[x_prev].right_neighbor=x_next;
          }
          fuse(x,x_next);
          x=x_next;
        }
      }
      x_next=container[x].right_neighbor;
    }
  }
}
```

Először összefésüli a kapott és a gyökérlistát $O(\lg n)$-es időben. Ez az INTERLEAVE eljárás feladata. Az eljárást a későbbiekben részletezem. Ezután végighalad a gyökérlistán. Az aktuális elemet x-ben, az előzőt x_prev-ben, a következőt pedig az x_next-ben tárolja. Kezdetben az x_prev értéke -1. Ha az aktuális elem fokszáma különbözik a következő elemétől vagy a következő elem nem -1, azaz létezik és a következő elem szomszédjának fokszáma egyenlő az aktuálisénval, akkor ugrik egyet a listán. Ha ezek nem teljesülnek, akkor további vizsgálat szükséges. Ha az x fokszáma kisebb, mint x_next-é, akkor összekapcsolja (FUSE) a két kupacot úgy, hogy x lesz a halom szülője. Ha nem kisebb, akkor fordított szereposztásban kell ugyanezt megtenni. Majd halad tovább, míg a lista végére nem ér.

Akkor most térjünk rá az INTERLEAVE-re, amely mint már említettem két kupac gyökérlistájának összefésülését végzi. Bemenő paramétere megegyezik a MERGE eljárás kapott paraméterével, azaz az összefésülendő kupac gyökérlistájára mutató fejelem (a). A head_other lesz az új összefésült lista fejeleme lesz, az other pedig ezen lista utolsó elemére mutat.

```
void interleave(int a) {
  int other=-1, head_other=-1;

  while( -1!=a || -1!=head ) {
    if( -1==a ) {
      if( -1==head_other ) {
        head_other=head;
      }
      else {
        container[other].right_neighbor=head;
      }
      head=-1;
    }
    else if( -1==head ) {
      if( -1==head_other ) {
        head_other=a;
      }
      else {
        container[other].right_neighbor=a;
      }
      a=-1;
    }
    else {
      if( container[a].degree<container[head].degree ) {
        if( -1==head_other )
        { head_other=a; }
        else {
          container[other].right_neighbor=a;
        }
        other=a;
        a=container[a].right_neighbor;
      }
      else {
        if( -1==head_other )
        { head_other=head; }
        else {
          container[other].right_neighbor=head;
        }
        other=head;
        head=container[head].right_neighbor;
      }
    }
  }
  head=head_other;
}
```

Három esetet kell meg különböztetni. Az első, amikor az a értéke NIL (-1), azaz a kapott gyökérlista üres, a második, ha a head értéke NIL (-1), a harmadik pedig, ha egyik sem üres. Az első esetben, ha a head üres, akkor az új lista a kapott lesz, különben pedig a meglevő listához adja a kapott lista „maradékát". A másodiknál ugyanezt teszi, csak fordított szereposztásban. A harmadik esetben viszont azon lista első elemét veszi, amelynek kisebb a kulcsa (prio értéke). Ekkor azt az elemet kiveszi (továbbléptetjük a fejelemet a testvérére), majd hozzáadja az új lista végére. Addig halad így, míg mindkét listából el nem fogynak az elemek. A végén az újonnan készített listára állítja a head fejelemet (head=head_other).

Az UNLACE végzi a minimum elem kiláncolását a gyökérlistából $O(1)$ időben. Mindössze végigmegy a gyökérlistán, megkeresi a kiláncolni kívánt elem bal

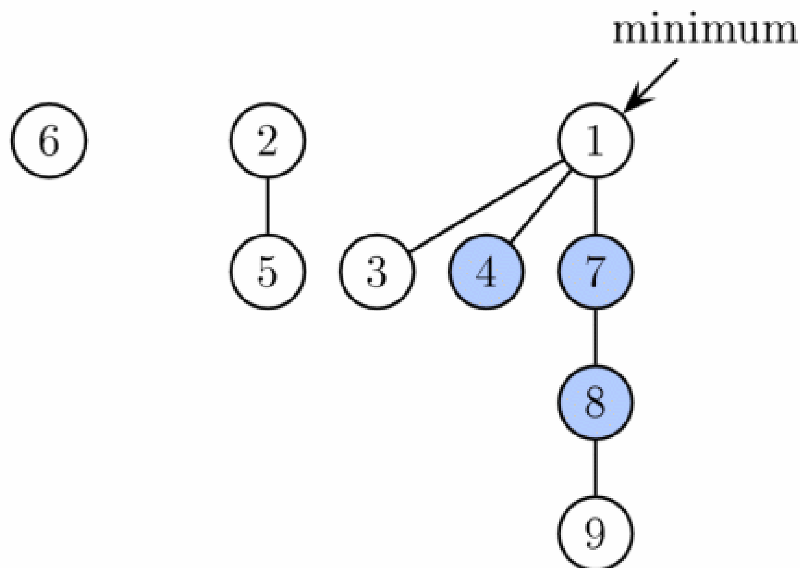
szomszédját, és a testvér pointerét (right_neighbor) a kapott csúcs jobb szomszédjára állítja.

A FUSE (összekapcsol) két B_{k-1} fát. A keletkezett B_k fa gyökércsúcsa az b lesz. Az eljárás $O(1)$ idő alatt megtehető. Azért ilyen egyszerű, mivel a kupac bal-gyerek jobb-testvérábrázolású. Elég, ha az új szülő legbaloldalibb gyerek lesz a másik halom. Ehhez be kell állítani a gyerek-szülő (parent-child) kapcsolatot és a testvér (right_neighbor) pointert.

3.1.3. FIBONACCI KUPAC

Korábban láthattuk, hogy a BESZÚR, MINIMUMOT-KERES, és az EGYESÍT műveletek $O(\lg n)$, valamint a KULCSOT-CSÖKKENT és a TÖRÖL műveletek binomiális kupacokon $\Theta(\lg n)$ legrosszabb futási időt adnak. Ennél a kupacnál látni fogjuk, hogy ha nem kell elemeket törölni, akkor ugyanezekre a műveletekre a sokkal jobb $\Theta(1)$ amortizált futási időt kapjuk. A Fibonacci kupacok különösen jól használhatók elméleti szempontból, ha a MINIMUMIT-KIVÁG és a TÖRÖL műveleteket kevesebbszer kell végrehajtani, mint a többi műveletet. Ez sok alkalmazásban is előfordul. Például, egyes gráf problémákat megoldó algoritmusok a gráfok minden élére csak egyszer hívják meg a KULCSOT-CSÖKKENT műveletet. Ha a gráf sűrű, akkor az $\Theta(1)$-es amortizált idejű hívások jelentősen javítják a bináris vagy binomiális kupacokra kapott $\Theta(\lg n)$ legrosszabb futási időt. Fő alkalmazásait azok az algoritmusok adják, amelyek mint a minimális feszítőfák számítása és az egy csúcsból induló legrövidebb utak megkeresésére alkalmaznak. Gyakorlati szempontból kevésbé előnyösek a legtöbb alkalmazásban, mint a bináris (vagy K-s) kupacok. Ha azonban több, a Fibonacci kupacokkal azonos amortizált időkorlátú egyszerűbb adatszerkezetet fejlesztünk, akkor gyakorlati hasznuk is van. A binomiális halomhoz hasonlóan ez a kupac is fákból áll. Ha a KULCSOT-CSÖKKENT és a TÖRÖL műveleteket egyszer sem hajtottuk végre, akkor a kupac minden fája olyan, mint egy binomiális fa. De a Fibonacci kupacoknak szabadabb szerkezetük van, és mindemellett aszimptotikus időkorlátokat tesznek lehetővé. A kupac karbantartását el lehet halasztani egészen addig, amikor az már könnyen elvégezhető.

A szerkezetének bemutatása a binomiális szerkezetére épül, és a Fibonaccira alkalmazott néhány művelet nagyon hasonló a binomiálisra alkalmazottakéhoz. A Fibonacci kupac is min-kupac-rendezett fák gyűjteménye.



Azonban a binomiális kupacokban levő fák rendezettek, ezzel ellentétben az ebben levő fák nem rendezettek, de van gyökérelemük. Minden x csúcsnak van egy *p[x]* pointere, amely a szülőjére, és egy *gyerek[x]* pointere, amely pedig az egyik gyerekére mutat. Az x csúcs gyerekei egy kétirányú ciklikus listával vannak összekapcsolva, ezt a listát az x gyerek-listájának nevezzük. A gyerek-lista minden y csúcsának van egy *bal[y]* és egy *jobb[y]* pointere, ezek az y baloldali illetve jobboldali testvéreire mutatnak. Ha az y egyedüli gyerek, akkor *bal[y]* = *jobb[y]* = y. A gyerekek sorrendje a gyerek-listában tetszőleges. A kétszeresen láncolt listák alkalmazásának két előnye is van. Az első, hogy egy elem kitörlése egy kétszeresen láncolt listából $\Theta(1)$ idő alatt elvégezhető. A második pedig, hogy két ilyen listát egy kétszeresen láncolt listába konkatenálni szintén $\Theta(1)$ időben lehel. Mindegyik csúcshoz még további két mezőt rendelünk. Az x csúcs gyerek-listájában levő gyerekek számát a *fokszám[x]* mezőben tároljuk. A logikai *megjelöl[x]* mező pedig jelzi, hogy az x elvesztette-e gyerekét, mióta x egy másik csúcs gyerekévé vált. Az újonnan létrehozott csúcsok nincsenek megjelölve, és egy x csúcs megjelöletlenné válik, amikor egy másik csúcs gyereke lesz. Egy adott Fibonacci kupac a minimális kulcsot tartalmazó fájának gyökérelemére mutató *min[H]* pointerrel adható meg, a minimális kulcsot tartalmazó csúcsot a kupac minimumcsúcsának nevezzük. Ha egy H kupac üres, akkor *min[H]* = NIL. A Fibonacci kupac fáinak gyökércsúcsai bal és jobb pointerekkel vannak összekapcsolva, ezt a ciklikus kétirányú listát a gyökérlistájának nevezzük. A fák sorrendje a gyökérlistában tetszőleges.

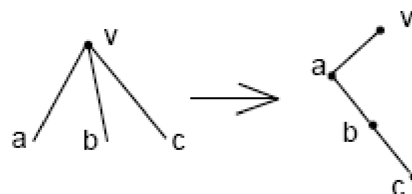
A binomiális kupacokhoz hasonlóan, ezt sem úgy tervezték meg, hogy bennük a KERES művelet hatékonyan elvégezhető legyen. A KUPACOT-KÉSZÍT művelet mind a binomiális mind a Fibonacci esetében $\Theta(1)$ idejű. A MINIMUMOT-KIVÁG futási ideje sem javult, azaz maradt $\Theta(\lg n)$-es. Ugyanakkor a MINIMUMOT-KERES, BESZÚR, KULCSOT-CSÖKKENT és az EGYESÍT műveletek $O(\lg n)$-ről $\Theta(1)$ futási idejűvé váltak. Ha n jelöli a kupacban szereplő elemek maximális számát, akkor bármely v elemnek legfeljebb n leszármazottja

lehet, azaz $r(v) \leq \lg_{\frac{\sqrt{5}+1}{2}} n \approx 1{,}44 * \lg n$. Tehát a Fibonacci kupaccal n darab beszúrás, n darab minimum-törlés, és e darab kulcs-csökkentés összes ideje $O(n * \lg n + e)$. Emiatt a Dijkstra-algoritmus futási ideje a Fibonacci-kupaccal $O(n * \lg n + e)$.

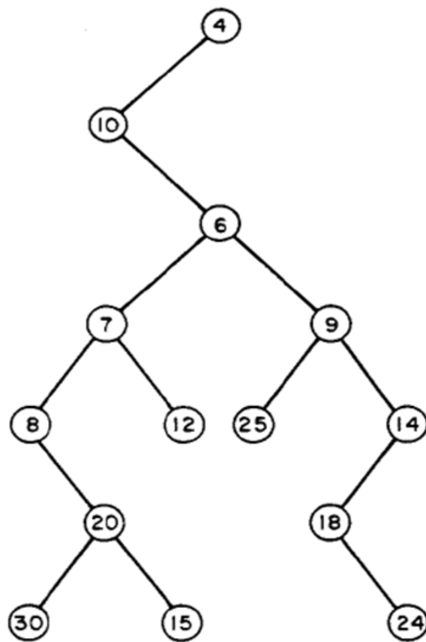
Ezt a kupacot már implementálták a LEMON könyvtár segítségével. Azért mutattam be mégis, mivel a párosítós kupac a Fibonacciból jött létre. Továbbá a tesztelésnél is hivatkozni fogok rá, amikor más kupacokkal hasonlítom össze.

3.1.4. PÁROSÍTÓS KUPAC

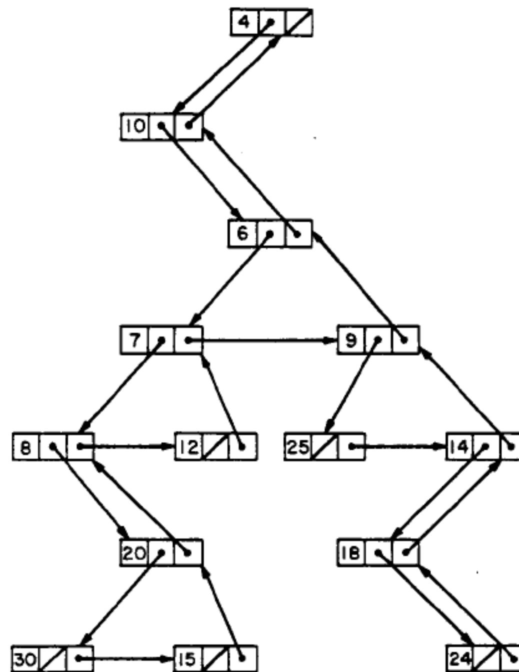
A párosítós-kupac létrejötte annak köszönhető, hogy a Fibonacci kupacnak is vannak hátrányai. Például sok pointert használ, emiatt nagy a helyigénye illetve nagy szorzók vannak, és az egy lépés is sok kis elemi műveletből áll. Az ötlet, hogy a Fibonacci kupacot akarjuk utánozni, de bináris fával és kevesebb pointerrel. Először emeljük fel a H gyökeret és helyére fűzzük be a gyerekeit. Majd forgassuk el 45°-kal az egészet, és ez lesz a párosítós kupac. Tehát itt a bal gyerek felel meg az első gyereknek, annak jobb gyereke pedig a következő testvérnek.



A párosítós kupac olyan bináris fát ábrázol, amely egyetlen gyökerének nem lehet jobb gyermeke. Így kapunk egy félig rendezett kupacot, azaz $K(v) \leq K(x)$ v bal fának minden x leszármazottjára. Így csúcsonként akár 2 pointert is nyerhetünk, azonban ha használjuk még a bináris fák memóriatakarékos megvalósítását is, akkor 2 pointert és 1 bitet. Tehát most az első pointer mutasson a bal gyerekre, ha létezik, ha nem akkor a jobb gyerekre, ha azonban ő sem létezik akkor a NIL-re. A második pointer pedig ha ő maga bal gyerek, akkor a testvérre mutat, ha létezik, különben a szülőre. Ha ő jobb gyerek akkor a szülőre mutat, de a gyökérben a NIL-re. Az 1 bit jelzi, hogy az adott gyerek bal gyerek-e. Gyakorlatban a párosítós kupac mindig jobb, mint a Fibonacci, de még nem találtak hozzá olyan potenciált, amivel be tudnák látni ugyanazokat az amortizációs időket.



Példa a párosítós kupac szerkezetére



Párosítós kupac pointeres ábrázolása

Az implementációban az előzőleg megadott leírást követtem, de a binomiális kupacnál már leírt módon ábrázoltam a pointereket. Röviden minden pointernek egy int típusú változó felel meg. Amelyik értékre mutat a pointer, a neki megfelelő változó azon érték a container nevű vektorban elfoglalt helyét tartalmazza.

```
class store {
  friend class PairingHeap;

  Item name;
  int parent;
  int child;
  bool left_child;
  int degree;
  bool in;
  Prio prio;

  store() : parent(-1), child(-1), left_child(false), degree(0), in(true) {}
};
```

Létrehoztam egy store nevű osztályt, amellyel reprezentálom a csúcsokat. Továbbá van egy friend osztálya, amire azért volt szükség, hogy a PairingHeap osztály is elérje az adattagokat. Minden csúcsnak van egy Item, amely a nevét, egy Prio, ami a kulcsát, és egy int adattagja, amely pedig a fokát fogja tartalmazni. Végül van még egy bool típusú adattagja is, amely megmondja, hogy az adott elem benne van-e a kupacban. Az első pointernek a child, a másodiknak a parent int típusú változó felel meg. Ezen kívül van egy logikai változó, amely megmondja, hogy az adott csúcs bal gyerek-e. Ha nem, akkor csak jobb lehet, mivel egy bináris fát ábrázol. A csúcsokat egy container nevű vektorban tárolom.

Akkor térjünk rá a BESZÚRÁS és a MINIMUMOT-KIVÁG műveletekre. Egy elem beszúrása a PUSH eljárással történik. Bemenő paraméterei egy Item és Prio típusú adatok. Először létrehoz egy egyelemű kupacot, majd meghívja a FUSE procedúrát, amely a két fa összelinkelését végzi. Mindenekelőtt az Item típusú változó alapján azonosítja a beszúrandó elemet, azaz meghatározza a vektorban elfoglalt helyét. Amennyiben nem szerepel az elem benne, akkor létrehozza, majd beszúrja a vektor végére. Ha már szerepel, akkor csak „alaphelyzetbe” hozza az adattagjait. Azért fordulhat elő, hogy egy elem szerepel a vektorban, de a fában nem, mivel amikor a fából egy elemet kitörlünk, akkor az a vektorból nem törlődik ki, csak az in nevű bool változó kap hamis értéket. A PUSH $O(1)$ időben elvégezhető, hiszen létre kell hozni egy egyelemű kupacot, majd össze kell linkelni a már meglevő kupaccal, amelyek $O(1)$ idő alatt elvégezhetőek. A POP művelet a minimális elemet törli ki a kupacból. A minimális elem is $O(1)$ időben megtalálható, hiszen a minimum nevű változó mindig tartalmazza a vektorban elfoglalt helyét.

```
void pop() {
   int TreeArray[num_items];
   int i=0, num_child=0, child_right = 0;
   container[minimum].in=false;

   if( -1!=container[minimum].child ) {
      i=container[minimum].child;
      TreeArray[num_child] = i;
      container[i].parent = -1;
      container[minimum].child = -1;

      ++num_child;
      int ch=-1;
      while( container[i].child!=-1 ) {
         ch=container[i].child;
         if( container[ch].left_child && i==container[ch].parent )
         {  i=ch;  }
         else {
            if( container[ch].left_child ) {
               child_right=container[ch].parent;
               container[ch].parent = i;
               --container[i].degree;
            }
            else {
               child_right=ch;
               container[i].child=-1;
               container[i].degree=0;
            }
            container[child_right].parent = -1;
            TreeArray[num_child] = child_right;
            i = child_right;
            ++num_child;
         }
      }
```

```
        int other;
        for( i=0; i<num_child-1; i+=2 ) {
           if ( !comp(container[TreeArray[i]].prio, container[TreeArray[i+1]].prio) ) {
              other=TreeArray[i];
              TreeArray[i]=TreeArray[i+1];
              TreeArray[i+1]=other;
           }
           fuse( TreeArray[i], TreeArray[i+1] );
        }
        i= (0==(num_child % 2)) ? num_child-2 : num_child-1;
        while(i>=2) {
           if ( comp( container[TreeArray[i]].prio, container[TreeArray[i-2]].prio) ) {
              other=TreeArray[i];
              TreeArray[i]=TreeArray[i-2];
              TreeArray[i-2]=other;
           }
           fuse( TreeArray[i-2], TreeArray[i] );
           i-=2;
        }
        minimum = TreeArray[0];
     }

     if ( 0==num_child )
     {  minimum = container[minimum].child;  }

     --num_items;
  }
```

Először levágja a gyökeret, majd az új gyökértől jobbra lefelé haladva minden élet elvág. Így sok kisebb párosítós kupac keletkezik. Tehát amíg létezik gyerek, addig halad a fán lefelé egy ciklus segítségével. Minden csúcsot megvizsgál, ha nincs gyereke, akkor beteszi a TreeArray tömbbe, majd végzett, de csak ha bal gyereke létezik, akkor a gyerek mentén halad tovább, ha azonban van jobb gyermeke is, akkor azt beteszi szintén a tömbbe a bal gyerekével együtt (ha van), majd a jobb gyerek mentén megy lejjebb. Amikor kivág egy elemet és beteszi a tömbbe, akkor tulajdonképpen egy részfát vág ki a kupacból. Így minden részfát eltárol. Ezek után össze kell linkelni (FUSE) a szétvágott kupacokat. Erre a legjobb megoldás, ha elindulunk lefelé és párosával összelinkeljük a kupacokat, majd ezután alulról felfelé haladva végzünk linkeléseket, mindig a következőt az előző eredményéhez. Azért ez a legjobb megoldás, mivel ekkor a két linkelendő kupacnak nagyjából megegyezik a mérete. A for ciklus végzi az előről-hátrafelé haladó párosával történő linkeléseket, majd a while ciklus pedig a visszafelé történőket. Minden linkelés előtt meg kell vizsgálni, hogy melyik gyökérelem kulcs értéke nagyobb, mivel a FUSE eljárás másodikként kapott kupacot illeszti az elsőként megadott alá. Amennyiben rossz sorrendben vannak a kupacok, akkor egyszerűen csak megcseréli őket a tömbben, így az eredmény mindig az előrébb levőben lesz. Legvégén pedig megadja az új minimumot. A minimum a TreeArray tömb legelső eleme lesz, tehát a legelső helyen levő részfa gyökéreleme, mivel a végén az első helyen levőt linkeli össze a második helyen levővel és az eredmény az előbbi helyén lesz. A pop műveletigénye $O(\lg n)$ a legrosszabb esetben, mivel végig kell menni a fán a gyökértől jobbra lefelé haladva a levélig. A fa magassága legrosszabb esetben $\lg n$.

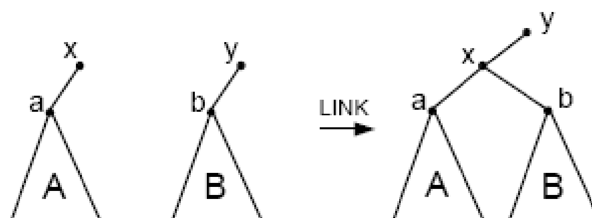
A Dijkstra-algoritmus szempontjából még ki kell emelni a DECREASE (kulcsot-csökkent) és az INCREASE (kulcsot-növel) műveleteket. A DECREASE eljárás egy adott elem kulcs értékét csökkenti.

```
void decrease (Item item, const Prio& value) {
  int i=iimap[item];
  container[i].prio=value;
  int p=container[i].parent;

  if( container[i].left_child && i!=container[p].child )
  {  p=container[container[i].parent].parent;  }

  if ( p!=-1 && comp(value,container[p].prio) ) {
    cut(i,p);
    if ( comp(container[minimum].prio,value) )
    {  fuse(minimum,i);  }
    else {
      fuse(i,minimum);
      minimum=i;
    }

  }
}
```

Az eljárás a kapott elemet kivágja a kupacból a baloldali részfájával együtt, majd összelinkeli az eredeti kupaccal. Tehát meghatározza az elem helyét a container-ben, majd a szülőjét. Ezután beállítja az új kulcs értéket és meghívja a CUT segédeljárást. Végül pedig a gyökérelemek kulcs értékeinek megfelelő paraméterezéssel meghívja a FUSE procedúrát. Amennyiben a kiválasztott elemnek nincs szülője, tehát gyökérelem, akkor elég csak beállítani az új kulcs (Prio típusú) értéket. A DECREASE ezen kupac esetében is $O(1)$ amortizációs idejű, hasonlóan a Fibonacci kupachoz, hiszen nem függ az elemek számától. Az INCREASE procedúra növeli a kiválasztott elem kulcs értékét, majd rendezi a kupacot. Ezt úgy valósítja meg, hogy először kitörli az elemet (ERASE) a kupacból, majd ismét beszúrja az új kulcs értékkel (PUSH). Az elem törlése a következőképpen történik. Meghívja a KULCSOT-CSÖKKENT eljárást a minimum értékénél 1-gyel kisebb értékkel és a kitörlendő elemmel. Így az elem felkerül a gyökérlistába és ő lesz az új minimum is. Ezek után csak meg kell hívni a POP műveletet.

Az előbb említett négy művelet elvégzéséhez használunk két segédeljárást a CUT-ot és FUSE-t. Ezeket a privát részben implementáltuk, hiszen a „külvilág", azaz például a Dijkstra-algoritmus számára érdektelenek.

A FUSE végzi két párosítós kupac összelinkelését. Két bemenő paramétere van. Mindkettő egy-egy párosítós kupac, amelyek gyökérelemükkel vannak megadva (int típusú változók, melyek tartalmazzák az elemeknek a container vektorban elfoglalt helyét). Továbbá tudja azt is, hogy az első kupac gyökéreleme kisebb, mint a másodiké, hiszen olyan sorrendben kapja meg a paramétereket. Így a másodikként kapottat kell az első alá linkelni.



Két kupacrendezett fa összelinkelése

```
void fuse(int a, int b) {
   int child_a = container[a].child;
   int child_b = container[b].child;
   container[a].child=b;
   container[b].parent=a;
   container[b].left_child=true;

   if( -1!=child_a ) {
      container[b].child=child_a;
      container[child_a].parent=b;
      container[child_a].left_child=false;
      ++container[b].degree;

      if( -1!=child_b ) {
         container[b].child=child_b;
         container[child_b].parent=child_a;
      }
   }
   else { ++container[a].degree; }
}
```

Első lépésként eltárolja a gyereket. Mindkét gyökérelemnek egyetlen baloldali gyereke van, hiszen a kapott részfák kisebb párosítós kupacok. A linkelés a fenti ábra alapján történik. Az első lesz az új kupac gyökéreleme (a), az ő bal gyereke lesz a másik gyökérelem (b). Amennyiben létezik az a-nak gyereke, akkor azt be kell illeszteni a b elem jobb oldali gyerekének. Ezt úgy oldottam meg, hogy először vizsgálom, hogy létezik-e az a-nak gyereke, mert ha nem, akkor csak az a-b elemek gyerek-szülő kapcsolatát kell beállítani. Ha van az a-nak gyereke, akkor azt a b-hez kell csatolni, és ráállítom a b elem child pointerét. Csak ezek után vizsgálom a b elem gyerekét. Ha létezik, akkor a b elem parent pointerét át kell állítani a b új gyerekére (child_a) és a child pointerét vissza kell állítani a régi gyerekére (child_b).

A másik segédeljárás a CUT szintén két bemenő paraméterrel. Az első paraméter a kivágandó elem, a második pedig a szülője.

```
void cut(int a, int b) {
  int child_a;
  switch (container[a].degree) {
    case 2 :
      child_a = container[container[a].child].parent;
      if( container[a].left_child ) {
        container[child_a].left_child=true;
        container[b].child=child_a;
        container[child_a].parent=container[a].parent;
      }
      else {
        container[child_a].left_child=false;
        container[child_a].parent=b;
        if( a!=container[b].child )
        { container[container[b].child].parent=child_a; }
        else
        { container[b].child=child_a; }
      }
      --container[a].degree;
      container[container[a].child].parent=a;
      break;

    case 1 :
      child_a = container[a].child;
      if( !container[child_a].left_child ) {
        --container[a].degree;
        if( container[a].left_child ) {
          container[child_a].left_child=true;
          container[child_a].parent=container[a].parent;
          container[b].child=child_a;
        }
        else {
          container[child_a].left_child=false;
          container[child_a].parent=b;
          if( a!=container[b].child )
          { container[container[b].child].parent=child_a; }
          else
          { container[b].child=child_a; }
        }
        container[a].child=-1;
      }
      else {
        --container[b].degree;
        if( container[a].left_child )
        { container[b].child= (1==container[b].degree) ? container[a].parent : -1; }
        else {
          if (1==container[b].degree)
          { container[container[b].child].parent=b; }
          else
          { container[b].child=-1; }
        }
      }
      break;

    case 0 :
      --container[b].degree;
      if( container[a].left_child )
      { container[b].child= (0!=container[b].degree) ? container[a].parent : -1; }
      else {
        if( 0!=container[b].degree )
        { container[container[b].child].parent=b; }
        else
        { container[b].child=-1; }
      }
      break;
  }
  container[a].parent=-1;
  container[a].left_child=false;
}
```

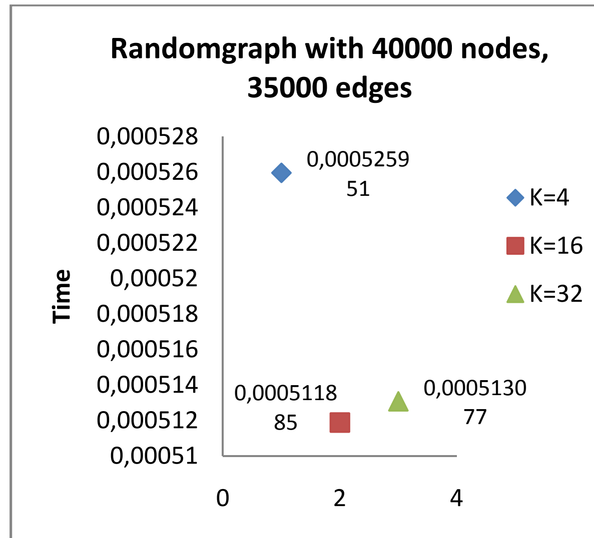
A kivágásnál hat eset lehetséges. Először is három, attól függően, hogy mennyi az adott csúcs foka. Majd mindegyiken belül, hogy az adott csúcs bal vagy jobb gyerek. Az első eset, amikor két gyereke van. Ekkor kivágja a csúcsot a bal részfájával együtt, és a jobb gyerekét pedig felviszi a helyére. Oda kell figyelni arra, hogy a jobb gyereket bal vagy jobb helyre kell felvinni. A különbség csak annyi, hogy más-más adattagokat kell átállítani. A következő eset, amikor csak egy gyereke van. Itt nem csak azt kell vizsgálni, hogy az adott csúcs bal vagy jobb gyerek-e, hanem azt is, hogy az ő egyetlen gyereke bal vagy jobb gyerek. Tehát itt további két eset lép fel. Először azt az esetet nézi, amikor a gyereke - amit a child_a változóba tárolt el - jobb oldali gyerek, azaz csak magát a csúcsot kell kivágni. A másik, amikor bal gyerek, ekkor őt is ki kell vágni a csúccsal együtt, azaz nem illeszt be semmit a kivágott csúcs helyére. Végül, ha nincs gyereke, akkor csak magát a csúcsot kell kivágni. Ilyenkor még arra is oda kell figyelni, hogy a szülőnek van-e még gyereke és eszerint kell beállítani az adattagokat. E két művelet szintén $O(1)$ idejű, mivel ezek sem függnek a csúcsok számától.

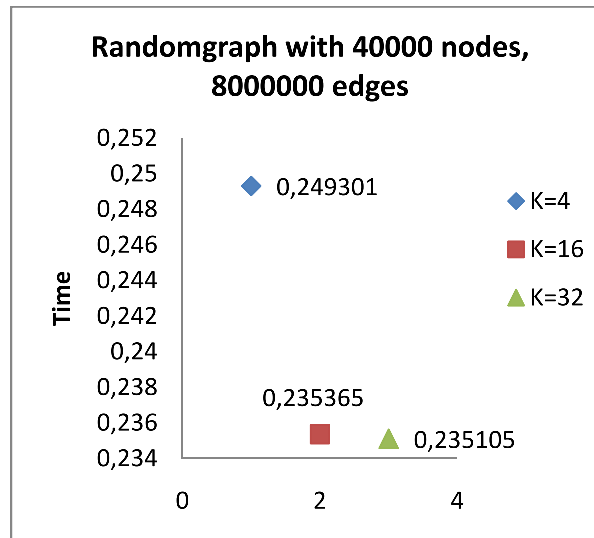
Tehát összefoglalva, a műveletek amortizációs ideje megegyezik a Fibonacci kupac amortizációs idejeivel. Ugyanakkor létezik többféle módszer, hogy lehet javítani a műveltek futási idejét. Továbbfejlesztési lehetőségként fenn áll ezen módszerek részletesebb megismerése és implementálása.

3.2. Tesztelés

Ebben a fejezetben foglalkozom az általam implementált és a már meglevő kupacokon futtatott Dijkstra-algoritmus futási idejeinek összehasonlításával különböző méretű és sűrűségű gráfokon. Főként a hasonló szerkezetű gráfokat hasonlítom össze. Így a binárist a 4-gyerekes és a K-gyerekes, a binomiálist pedig a párosítós és a Fibonacci kupaccal. A K-s kupac fajtáit mindig K=x-szel fogom jelölni, ahol az x nyilvánvalóan a gyerekek száma lesz.

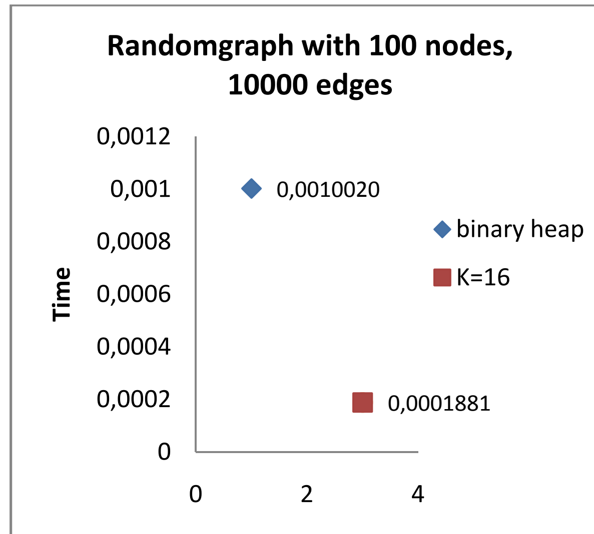
3.2.1. A K-gyerekes kupac tesztelése K = 4,16,32 esetekben

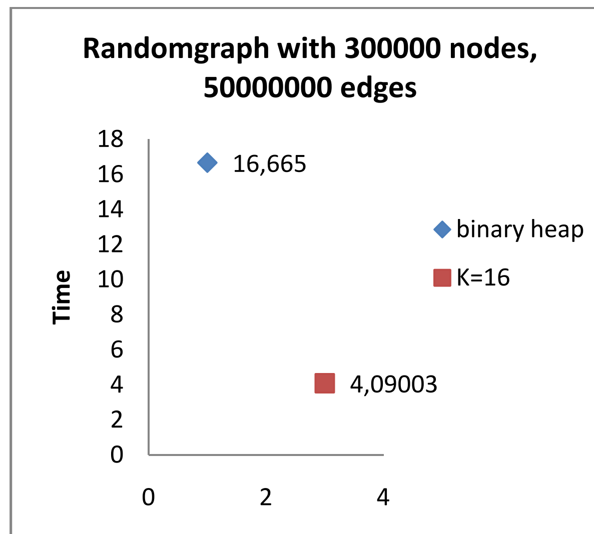




A fenti két diagramból látható illetve több további példával is illusztrálható, hogy általában a Dijkstra-algoritmus a legjobb futási idejét a K=16 gyerekű kupacon éri el. Ugyanakkor a legrosszabb időt a K=4 gyerekes kupacon produkálta. Ez azért is meglepő, mivel a későbbiekben látható, hogy a 4-ary kupac szinte mindig az egyik legjobb időt érte el. A különbség az implementációban van, hiszen a 4-gyerekes kupac minimum-kereső függvénye négy gyerekre lett optimalizálva, míg K-gyerekes esetében ezt nem tehettük meg, mivel a K tetszőleges értéket vehet fel.

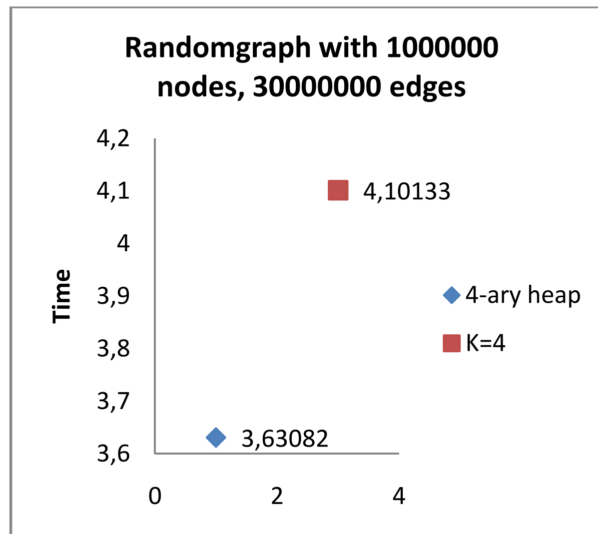
3.2.2. A bináris és a K=16-gyerekes kupac összehasonlítása





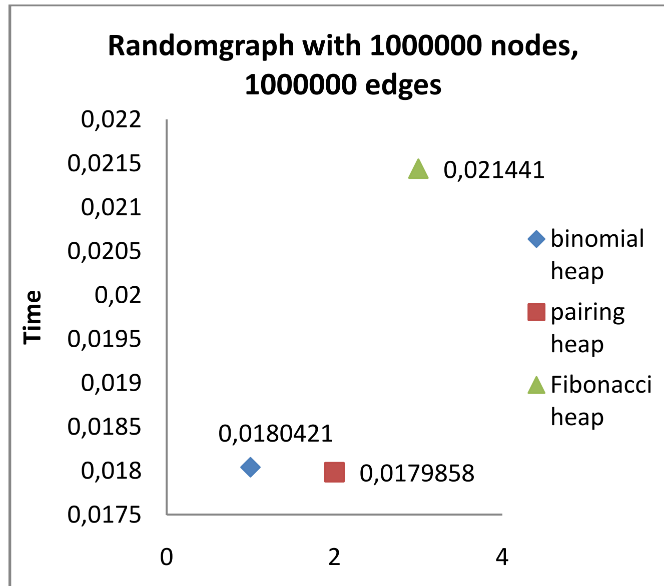
Látható, hogy nem csak nagy gráfok esetében, hanem már kicsi mindössze 100 csúccsal és 10000 éllel rendelkező gráf esetében is sokkal kedvezőbb futási időt produkál az algoritmus a 16-gyerekes kupac esetében. Ugyanakkor érdekes, hogy nagy sűrűségű gráfok esetében 3-5-szöröse, míg kis sűrűségűeknél jóval nagyobb a különbség arányaiban. Ekkor általában 15-20-szorosa a bináris futási ideje. Persze ezek az adatok azért függnek a gráfok típusaitól is, de az általam tesztelt összes gráfon sokkal rosszabb futási időt ért el a bináris halom.

3.2.3. A 4-gyerekes és a K=4-gyerekes kupacok összehasonlítása



Ebben az esetben két azonos adatstruktúrát vizsgálunk, mégis ehhez képest jelentős különbséget vélhetünk felfedezni a teszteredményekben. Látható, hogy ezen a nagyobb gráfon már majdnem fél másodperc az eltérés. Gondolhatnánk, hogy ez nem is olyan nagy, de általában egy alkalmazásban nem egyszer, hanem sokkal többször fut le a Dijkstra-algoritmus. Így összességében ez a sok kicsinek tűnő különbség kiadhat már egy elég jelentős időkülönbséget is. Amint már korábban említettem, ez az eltérés vélhetően a minimum-kereső függvény implementációjában rejlik. A K-gyerekes kupac esetében általánosan kellett megvalósítani, ezzel szemben a 4-gyerekesnél lehetőségünk volt a kódot nagyobb mértékben optimalizálni. Tehát az elméleti hatékonyság mellett, az implementáció hatékonyságát is gyakran vizsgálni kell.

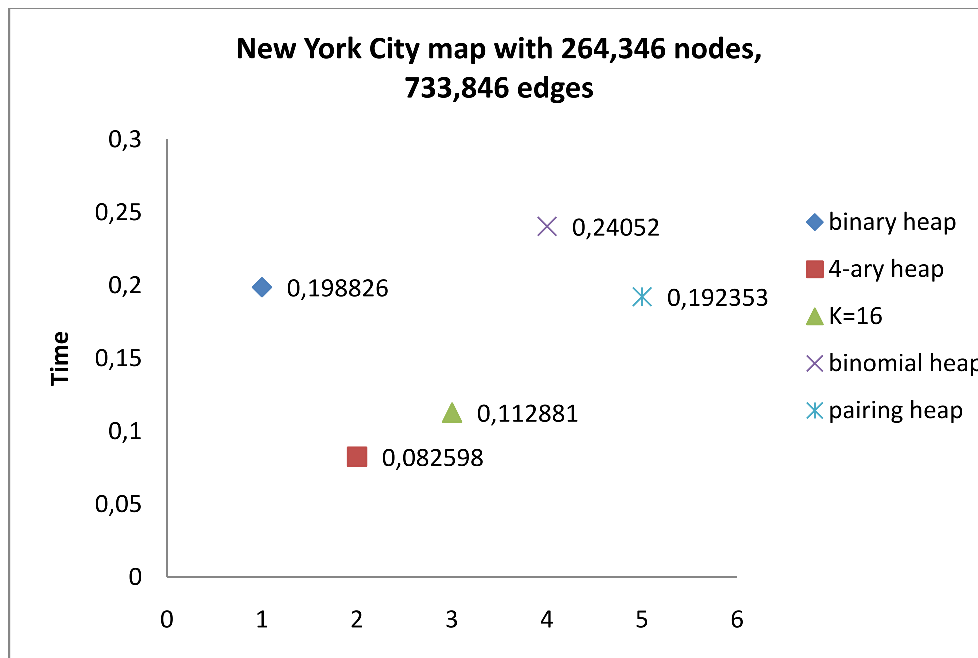
3.2.4. A binomiális, párosítós és a Fibonacci kupac tesztelése



Míg az előzőekben általában a „győztes időt” nem befolyásolta a gráfok nagysága, sűrűsége, addig e halmok esetében már fontos szerepet tölt a gráf sűrűsége. A fenti ábrán egy ritka gráf látható, és leolvasható, hogy a legjobb időt a párosítós kupac érte el. Ez a „szabályosság” további ritka gráfok esetében is felfedezhető függetlenül a gráfok csúcsainak a számától. Ugyanakkor sűrű gráfok esetében meg a Fibonacci kupacon futtatott Dijkstra-algoritmus futási ideje a legkisebb. A binomiális kupacon az általam tesztelt gráfokon általában az előbb említett két kupac futási ideje között helyezkedik el.

3.2.5. New York úthálózatán futtatott Dijkstra-algoritmusok

Ne csak véletlen gráfokon végezzek teszteléseket, ezért New York város úthálózatán, mint gráfon is futtattam az algoritmus. Ez a gráf körülbelül 250 000 csúcsot és 750 000 élet tartalmaz. Tehát egy ritka gráfról van szó.



Az eredmények megegyeznek a korábban tapasztaltakkal. A legjobb idő a 4-gyerekes kupacé, aztán K=16-gyerekes kupac is elég jól szerepelt, megelőzte a párosítós kupacot is. Érdekesség, hogy a bináris kupac ideje kevesebb, mint a binomiálisé, ennek oka talán a megvalósításban található.

3.2.6. Értékelés

Az általam folytatott tesztelésekből az derül ki, hogy általánosan a legjobban szereplő kupacok a 4-gyerekes és a K-gyerekes kupacok azon belül is a K=16. Ha sok gráfon kellene folytatnom teszteléseket, akkor e két kupac közül választanék. Azonban ha a gráf éleinek a száma elég közel van a gráf csúcsainak a számához, azaz a gráf eléggé ritka, akkor a párosítós kupacot választanám. Néhány speciálisabb gráf esetében választanám csak a Fibonacci kupacot.

4. Összefoglalás

Mint a szakdolgozatomban láthattuk a 4-gyerekes, K-gyerekes, binomiális és párosítós kupacokat valósítottam meg. Igyekeztem egy általános az alapoktól kezdődő bevezetés után mindegyik kupacot részletesen leírni, hogy annak implementálása könnyen kezelhető legyen. Ebben nagy segítségemre volt az Cormen-Leiserson-Rivest-Stein: Introduction to Algorithms könyvének magyar fordítása. Próbáltam egyszerű, könnyen érthető kódot implementálni, ugyanakkor nem eltérni a LEMON-ban használt konvencióktól. A teszteléseknél odafigyeltem, hogy véletlen gráfok mellett, valós gráfokon is teszteljem a kupacokat. Az általam kapott eredményeket összefoglaltam pár mondatban, a könnyebb érthetőség kedvéért.

Mivel a témát érdekesnek találtam, ezért további célom, hogy e kupacok futási idejeit javítsam, azaz a kódokat még jobban optimalizáljam, illetve új gyorsabb ötleteket valósítsak meg bennük. Továbbá új kupacokat is szívesen megvalósítanék.

Irodalomjegyzék

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest: Algoritmusok, Műszaki könyv Kiadó, Budapest, ISBN: 963 16 3029 3

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein: Új Algoritmusok, Scolar Kiadó, Budapest, ISBN: 963 9193 90 9

Robert Endre Tarjan: Data Structures and Network Algorithms, SIAM kiadó, Philadelphia, ISBN 0-89871-187-8

Daniel Dominic Sleator, Robert Endre Tarjan: Self-Adjusting Heaps, SIAM Journal on Computing, Volume 15, Number 1, 1986

Michael L. Fredman, Robert Sedgewick, Daniel Dominic Sleator, Robert Endre Tarjan: The Pairing Heap: A New Form of Self-Adjusting Heap. Algorithmica, Volume 1, Number 1, 1986

Aszalós László: Algoritmusok, Első kiadás, MobiDiák könyvtár, Debreceni Egyetem Informatikai Kar, 2004

Király Zoltán: Adatstruktúrák. 2.31-es verzió. 2008. május 3.
http://www.cs.elte.hu/~kiraly/Adatstrukturak.pdf

A LEMON ELTE-s oldala. 2008. március 8.
https://lemon.cs.elte.hu/site/

Bináris kupac. 2008. április 5.
http://en.wikipedia.org/wiki/Binary_heap

Binomiális kupac. 2008. április 5.
http://en.wikipedia.org/wiki/Binomial_heap

Fibonacci kupac. 2008. április 5.
http://en.wikipedia.org/wiki/Fibonacci_heap

9th DIMACS Implementation Challenge - Shortest Paths. 2008. május 3.
http://www.dis.uniroma1.it/~challenge9/download.shtml

Köszönetnyilvánítás

Először is szeretném megköszönni szüleimnek, keresztszüleimnek, nagyszüleimnek, hogy eljuthattam idáig, és diplomázhatok. Sok-sok bátorítást és noszogatást is kaptam tőlük az életben. A szakdolgozat elkészítésében nagyon sokat segített témavezető tanárom Dezső Balázs, illetve Kovács Péter, akik nagy türelemmel és segítőkészséggel voltak irántam. Bármikor bármilyen kérdéssel fordulhattam hozzájuk. Nekik is nagyon köszönöm. Végül köszönöm diáktársaimnak, barátnőmnek, barátaimnak, az egyetemnek, hogy mindig számíthattam rájuk.