EÖTVÖS LORÁND TUDOMÁNYEGYETEM
INFORMATIKAI KAR
INFORMÁCIÓS RENDSZEREK TANSZÉK

Konvex- és mozgó objektumok indexelése

Diplomamunka

Témavezető:
Dr. Benczúr András
egyetemi tanár
ELTE IK

Készítette:
Orgován Krisztina
programtervező matematikus szak
NAPPALI tagozat

Budapest
2011

Tartalomjegyzék

1. Bevezetés

Számos közismert példa mutatja meggyőzően, hogy napjainkban egyre nő a helyhez kötött információ jelentősége. A „hely” általában síkon értelmezhető, gyakran azonban a domborzati magasságokkal is számolni kell, olykor pedig az idő, mint további dimenzió egészíti ki az előzőket. A térinformatika szerepe a társadalmi, gazdasági, mezőgazdasági, környezet- és katasztrófavédelmi, valamint településfejlesztési összefüggésekben meghatározó, és alkalmazási területei rohamosan bővülnek. A térinformatika eredményeinek magáncélú felhasználása is egyre számottevőbb, például a tájékozódás hardver- és szoftvereszközeinek igénybevételével.

A geoinformatikai alkalmazások mindegyikében igen nagy mennyiségű adat hatékony tárolását és kezelését szükséges megoldani. Az egyre bonyolultabb feladatok megoldása és az egyre kifinomultabb algoritmusok implementálása hatékony indexelési módszerek megválasztását teszik szükségessé. Diplomamunkám alapvető célkitűzése a legismertebb térbeli indexelési eljárások összegyűjtése és bemutatása. Választásomat személyes érdeklődési köröm is motiválta. Egyetemi tanulmányaim során leginkább három terület került közel hozzám; amelyek a következők, az algoritmusok és adatstruktúrák elmélete, az adatbáziskezelés, illetve általánosabban az információkezelés világa, valamint a térinformatika alkalmazásai. Témaválasztásom kreatív módon ötvözi ezt a három területet.

A következő, második fejezetben áttekintem a diplomamunkám további részeinek megértéséhez szükséges alapvető ismereteket. Az adatbáziskezelés alapfogalmai után főként az indexelés lehetséges módszereit és felhasználási céljait mutatom be átfogó jelleggel. Az indexelések közül részletesebben a B-fával foglalkozom, mivel a további fejezetekben használt struktúrák alapvetően erre épülnek.

A harmadik fejezetben rátérek a térbeli adatok tárolására. Először is érintem a raszteres és vektoros adattípust, amelyek alapvetően eltérő kezelési módot igényelnek. Dolgozatomban – néhány példától eltekintve – a vektorizált adatok állnak a vizsgálat előterében. Hosszabban foglalkozom ezután a térbeli indexelések bemutatásával, kiemelve az egydimenziós indexeléstől való eltéréseket illetve a hasonlóságokat is. Az indexelő eljárásoknak két típusát vezetem be, amelyeknek módszereit a következő két fejezetben mutatom be részletesen. A szakirodalmakban többféle osztályozással találkoztam, én mégis ezt a felosztást tartom követendőnek.

A negyedik fejezetben az első csoportba tartozó módszerekkel foglalkozom részletesen. A közös jellemzőjük, hogy mindegyik indexelés a keresési teret osztja fel különböző alakú és méretű területekre. Ezen tulajdonság alapján, ebbe az osztályba soroltam a grid index két típusát, a fix gridet és a grid-file-t (dinamikus grid), a kd-fát és annak egy gyakrabban használt változatát, az adaptív kd-fát, a kd-fa és a B-fa egy lehetséges ötvözetét, a kd-B-fát és a BSP-fát. Ebbe a fejezetbe ismertetem a térkitöltő vonalak két ismert változatát a Hilbert-görbét és a Z-vonalat. Ezek ismeretében térek rá a negyedelő-fára és annak három – felhasználástól függő – fajtájára, a lineáris negyedelő-fára, illetve a Z-rendező fára. Mindegyik struktúra esetén előtérbe helyezem a pont- és ablaklekérdezés algoritmusait, az alfejezetek végén pedig röviden érintem a hatékonysági tulajdonságaikat.

Az ötödik fejezetben áttérek az adatvezérelt módszerekre, amelyek nem a térből, hanem az abban elhelyezkedő objektumokból indulnak ki, és valamilyen téglalap közelítéssel szervezik az alakzatokat egy adatszerkezetbe. Ide soroltam a térinformatikában leggyakrabban használt R-fát és annak a családjába tartozó R^+-fát és R^*-fát. Mindhárom esetben részletesen leírom a hozzájuk tartozó keresés, törlés és beszúrás algoritmusait. Az R-fák témakörét a költség modelljük elemzésével zárom. Ebben a fejezetben kap helyet az a mélyebb, önálló munkát igénylő ismertetés, amely a szegmensfa és intervallumfa beható tárgyalása után a konvex alakzatok indexelésére tér át. A szegmensfa az ún. sík söprés (*plane sweeping*) módszerét támogatja, és az alakzatok befoglaló téglalapjával kapcsolatban olyan alapvető kérdések hatékony megválaszolását teszi lehetővé, mint a pont tartalmazás, illetve a metsző szegmens- és téglalappárok meghatározása.

Diplomamunkám számomra legizgalmasabb része, a konvex alakzatok indexelése, következik ebben a fejezetben. Abból a feltételezésből indulok ki, hogy trapézokra bontással egy olyan vegyes indexelést lehet megvalósítani, amellyel a korábban osztatlan alakzatokat a részekre osztás finomságával lehet kezelni bizonyos lekérdezések megválaszolása során. Kiderült azonban, hogy a trapézokhoz nem minden esetben illeszkedik jól a szegmensfa index illetve a mögötte álló sík söprés elve. Ehelyett a másodlagos indexeléshez használt kiegyensúlyozott bináris keresőfákból indultam ki, vagy pedig a szeletekre bontást követően a létrejött trapézok befoglaló téglalapjaira kellett áttérni.

A hatodik, záró fejezetben foglalkozom a térinformatikai tárolás egy speciális esetével, amelyben az idő, mint új dimenzió játszik szerepet. Az egyik legalapvetőbb

indexelés mozgó objektumok esetén a TPR-fa, amely a fejezet jelentős részét képezi. A TPR-fa alfejezet után a költség modelljét elemzem, majd áttérek a másik mozgó alakzatokat kezelő adatszerkezetre. a parametrikus R-fára. Az utóbbiak alkalmazásakor nem csak az idő, hanem az alakzatoknak egy ún. sebesség vektora is meghatározó. A fejezet legvégén egy gyakorlati alkalmazásra mutatok példát.

Dolgozatom végén összefoglalom a munkám legfontosabbnak tekinthető eredményeit.

Igyekeztem mondanivalómat sok szemléletes ábrával illusztrálni. A csaknem 60 ábra közül számosat magam rajzoltam meg, egy része pedig az irodalomjegyzékben szereplő forrásokból származik.

A dolgozatot, a több mint 20 indexelési módszer keletkezésének kronológiája, ábrajegyzék és irodalomjegyzék zárja.

2. Az indexelés fogalma

Ebben a fejezetben áttekintem a diplomamunkám többi fejezetéhez szükséges alapvető ismereteket. Az összefoglalás nem a legnagyobb részletességgel készült, hanem inkább átfogó jellegű. Csak a B-fa indexelés kerül leírásra, de ezen kívül természetesen léteznek egyéb eljárások is. Azonban ezek az egyéb indexelések, a dolgozatban ismertetésre kerülő térbeli indexelési módszereknél, nem játszanak szerepet.

2.1 Adattárolás és adatkezelés

Egy adatbázis-kezelő rendszer (*DBMS – Database Management System*) bármely formája két fő funkciót kell, hogy ellásson. Támogatnia kell nagyon nagy mennyiségű adatok tárolását, emellett a tárolt adatok hatékony és gyors kezelését. Adatkezelésen általában az adateléréseket illetve a rajtuk való különböző műveletvégzéseket értjük. Az adatokon végzett leggyakoribb műveletek a különböző szempontok alapján való lekérdezések (*SELECT*) mellett, a beszúrás (*INSERT*), a törlés (*DELETE*) és a módosítás (*UPDATE*).

Az adatkezelések a központi memóriában történnek. A memória véletlen hozzáférésű, azaz bármelyik bájt elérési ideje közel ugyanakkora.

A rendszer egyik legfontosabb jellemzője, hogy az adatok tárolására a központi memória mellett másodlagos adattárolókat használ, más néven háttértárolókat. Ez a tárolóknak olyan formája, amely lényegesen lassabb, de ugyanakkor sokkal nagyobb kapacitású, mint a központi memória. Manapság a másodlagos tárolás szinte kizárólag mágneses lemezeken alapul. A lemezek teljes szerkezetének részletes ismertetését itt nem adom meg, a téma megértéséhez elég lesz pár alapvető tulajdonság kimondása.

Egy lemez fix méretű blokkokra van felosztva. A lemez blokkjainak mérete 4 KB és 56 KB között mozog. Például az Oracle adatbázis-kezelő rendszer alapértelmezésben 8 KB-os blokkokat használ. Az adatfájlok, nevezzük őket relációknak, ilyen blokkokban helyezkednek el a háttértárolón. Egy reláció rekordjai több blokkot is elfoglalhatnak. Adatok olvasásakor és írásakor a lemez és a központi memória között az adatfájlok, vagy azoknak részei, blokkok formájában mozognak (virtuális memória). Egy blokk jelenti a legnagyobb, egyszerre átvihető egységet. A központi memóriában egy blokknak egy lap felel meg. A blokkmozgatásokért az adatbázis-kezelő rendszer saját maga a felelős. Az adatmozgásokat röviden I/O műveleteknek nevezzük. Egy

adatmozgatás minden esetben költséggel jár. A feltételezés az, hogy az I/O műveletek költsége arányos a központi memória és a háttértároló között mozgatott blokkok számával. Mint azt az előzőekben említettem, az adatkezelések a központi memóriában mennek végbe, az adatok tárolása pedig egy másodlagos tárolón történik, így minden esetben, mikor adatokat kérdezünk le, vagy adatokat módosítunk, közben legalább két I/O műveletet is végrehajtódik. Egy a háttértárolóról a központi memóriába való adatolvasáskor, egy pedig a központi memóriából a lemezre való visszaíráskor. Az adatbázis-kezelők számára ezek a blokkmozgatások jelentik a legnagyobb költségeket. Egy I/O műveletigénye nagyságrendekkel nagyobb, mint a központi memóriában az adatokon végzett műveletek költsége. Az utóbbiaké olyan kicsi, hogy jelen esetben el is hanyagolható.

Akár egy, akár több blokkban tárolt reláció esetén, ha a reláció soraira vagyunk kíváncsiak, meg kellene vizsgálni a háttértároló összes blokkját, illetve azoknak, és a bennük lévő rekordoknak a fejléceit, hogy megtudjuk azt, hogy a kérdéses reláció sorai mely blokkokban helyezkednek el.

Egy példa SQL lekérdezés a *Személyek(TAJszám, név, cím, szüldat, életkor)* táblából:

SELECT * FROM Személyek;

Az ilyen típusú lekérdezéseknél láthatjuk, hogy nem jó megoldás, ha egy reláció sorait reprezentáló rekordokat szétszórva, a háttértároló különböző blokkjaiban helyezzük el. A kereséskor túl sok I/O műveletet kellene végrehajtani ahhoz, hogy az összes blokkot átvizsgáljuk. A megnövekedett műveletigény a keresést nagyon lelassítja.

Ha előre, a rekordok beszúrásának megfelelő sorrendben, lefoglalunk blokkokat egy reláció számára, akkor a keresett rekordokat tartalmazó blokkokat könnyedén megtalálhatjuk az egész háttértároló átvizsgálása nélkül is. Azonban ez továbbra sem segít, ha a reláció egy olyan sorára vagyunk kíváncsiak, melynek egyik mezője megegyezik egy előre megadott értékkel. Ilyen feltételes SQL lekérdezés lehet például:

SELECT * FROM Személyek WHERE név = 'Béla';

Egy feltételes lekérdezés eredményének meghatározásához végig kellene nézni minden olyan blokkot a háttértárolón, melyben a *Személyek* reláció sorait reprezentáló rekordok megtalálhatók és ezek közül kikeresni azokat, amelyekre a *név = 'Béla'* feltétel teljesül. A keresés még mindig nagy költségekkel járna az I/O műveletek száma miatt.

Az egyik cél tehát, a rekordokból álló adatállomány fizikai tárolása a blokkokban úgy, hogy az adatokhoz való későbbi hozzáférés minél gyorsabb legyen. Erre több ismert módszer is létezik. A tárolt adatok gyors elérése azonban nem kizárólag a tárolás módszerétől függ. Itt kap jelentőséget az indexelés fogalmának bevezetése, és az, hogy egy index hogyan gyorsítja meg a kereséseinket.

2.2 Indexelés

Az előző alfejezetben említett feltételes lekérdezések hatékony végrehajtásához gyakran hozunk létre a relációkon egy vagy több indexet. Az index egy olyan adatszerkezet, melynek segítségével gyorsan megtalálhatjuk azokat a rekordokat, amelyek adott tulajdonsággal rendelkező mezőket tartalmaznak. Az említett tulajdonság egy rekord egy vagy több mezőjének az értékére vonatkozik. Az index lehetővé teszi, hogy kereséskor a rekordoknak csak egy kisebb halmazát kelljen megvizsgálni. Minél kisebb a megvizsgálandó halmaz, annál kevesebb költséggel jár a művelet elvégzése.

Indexet egy adattábla egy vagy több oszlopára hozhatunk létre. Egy index alapjául szolgáló oszlopot keresési kulcsnak nevezünk. A keresési kulcs nem feltétlenül egyezik meg a reláció bármely más kulcsával, illetve nem szükséges, hogy egyedi legyen, azaz egy kulcsérték többször is szerepelhet. Továbbá az sem elvárás, hogy a táblázat bármely index attribútumára nézve rendezett legyen.

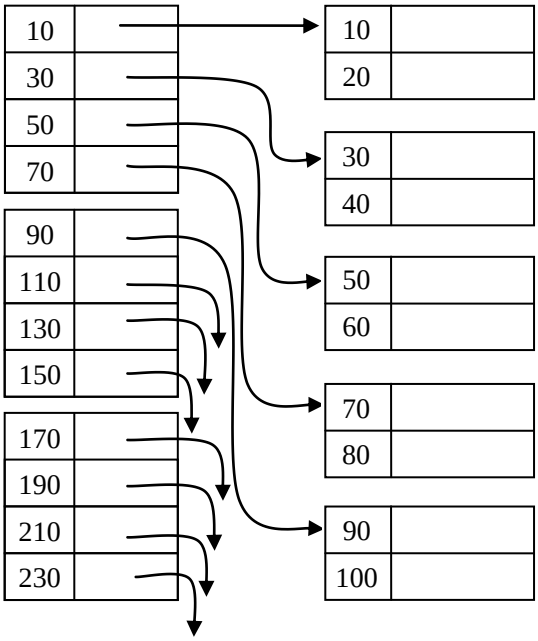
A legegyszerűbb index kulcs és mutató párokból áll. A kulcsok értékei megegyeznek a reláció indexelt oszlopának értékeivel rendezett sorrendben, független attól, hogy maga az adattábla az index attribútuma szerint rendezett volt-e vagy sem. Egy kulcshoz tartozó mutató az adattáblában arra a rekordra mutat, amely az adott kulcsértéket tartalmazza. Ezek a kulcs-mutató párok szintén blokkokba szerveződnek, amiket indexblokkoknak nevezünk. Egy index tehát tekinthető ilyen indexblokkok sorozataként is.

Ha az indexben az adattábla keresési kulcsának minden értéke szerepel, akkor **sűrű indexről** beszélünk. Ha a relációnak csak néhány rekordjához tartozik bejegyzés az indexben, akkor azt **ritka indexnek** nevezzük. Ebben az esetben az adatfájl minden blokkjához egy indexbejegyzés tartozik, így az indexrekordok száma egyenlő az adatfájl blokkjainak számával.

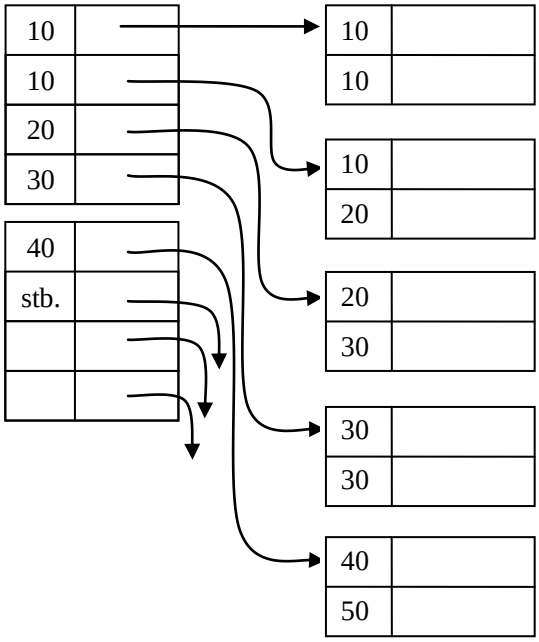
Az indexek között további különbséget jelent, hogy **elsődleges** vagy **másodlagos**-e az index. Elsődlegesnek nevezzük azokat az indexeket, amelyek meghatározzák az

indexelt rekordok helyét, azaz, az adatfájl a keresési kulcs attribútumára nézve rendezett (szekvenciális fájl). Tipikus eset, mikor egy adattábla egy oszlopára elsődleges kulcsot hozunk létre, amivel automatikusan létrejön az oszlopra egy index is. Jellemző, hogy az adatbázis-kezelő rendszerek az adattábláikat az elsődleges kulcsuk szerint rendezve tárolják, így a létrejött index is elsődleges lesz. Egy relációra csak egy elsődleges index hozható létre, mert egy indexelt tábla egyszerre csak egy attribútumára nézve lehet rendezett. Egy elsődleges index lehet sűrű is és ritka is.

Ha az elsődleges index ritka és a **keresési kulcsok egyediek**, akkor az adatfájlhoz tartozó blokkok legelső és egyben legkisebb kulcsértékű rekordjához tartozik csak indexbejegyzés (2.1 ábra). **Nem egyedi kulcsokra** létrehozott, ritka, elsődleges index használatakor egy jó megoldás, ha az index az adatfájl minden blokkjának legkisebb, új előfordulású kulcsértékéhez tárolunk egy indexrekordot (2.2 ábra). Ha egy blokkban nincs az előző blokkokhoz képest új előfordulású érték, akkor a legelsőre mutat az indexünk mutatója.



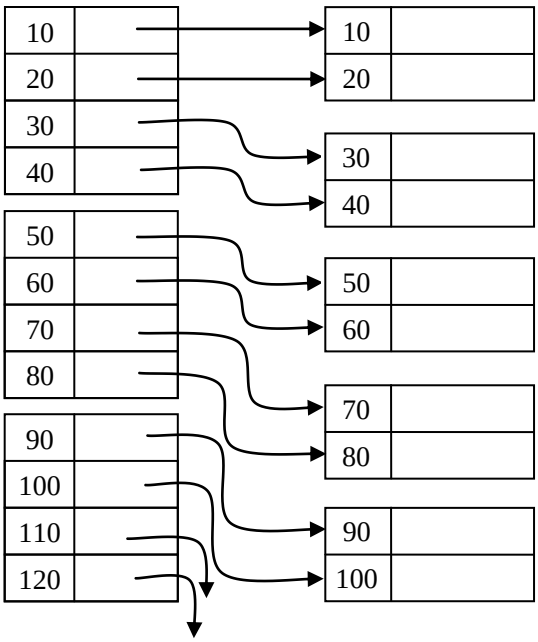
2.1 ábra: Ritka elsődleges index egyedi keresési kulcsokra



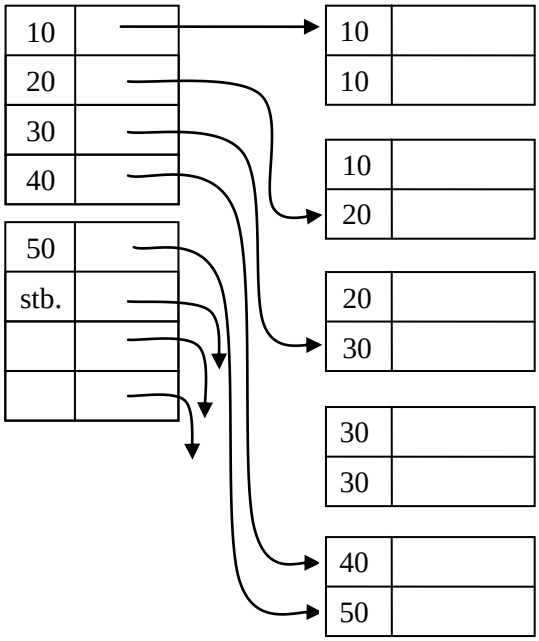
2.2 ábra: Ritka elsődleges index nem egyedi keresési kulcsokra

Egyedi kulcsokra létrehozott, sűrű, elsődleges index használatakor, a megadott definíció szerint, minden adatrekordhoz tárolunk egy indexrekordot (2.3. ábra). Abban az esetben, ha a **kulcsértékek nem egyediek**, a sűrű index méretét tovább csökkentheti, ha a reláció azonos kulcsértékeihez az indexben csak egy rekordot tárolunk, amelynek mutatója az adott kulcsérték első előfordulására mutat (2.4. ábra). A fájlunk

rendezettsége miatt, az előbbiek alapján már könnyedén megtalálhatjuk a további rekordokat is.



2.3 ábra: Sűrű elsődleges index egyedi keresési kulcsokra



2.4 ábra: Sűrű elsődleges index nem egyedi keresési kulcsokra

Előfordulhat azonban, hogy a gyorsítás érdekében az adattáblán további indexek létrehozására is szükségünk van. Tekintsük ehhez az alábbi SQL lekérdezést:

SELECT TAJszám, név FROM Személyek WHERE szüldat = '1983-11-12'

A *Személyek* reláció elsődleges kulcsának megfelel a *TAJszám* oszlop, mert az abban szereplő értékek biztos, hogy egyediek, hiszen nincs két különböző személy, akiknek a TAJ száma ugyanaz lenne. Feltételezzük, hogy a táblázatunk a *TAJszám* oszlopa szerint növekvően rendezett. Azáltal, hogy elsődleges kulcsnak tekintjük ezt az oszlopot illetve a reláció rendezettsége miatt, azonnal kaptunk is egy elsődleges indexet. Mi azonban most szeretnénk, az SQL lekérdezésben szereplő, *szüldat = '1983-11-12'* feltételnek megfelelő rekordok keresését felgyorsítani. Szükségünk lesz még egy indexre a reláció *szüldat* attribútumán is. Ez egy másodlagos index lesz. Egy relációra több másodlagos index is létrehozható, de másodlagos index kizárólag sűrű lehet. Ritka esetről nincs értelme beszélni, hiszen az adattábla a másodlagos index attribútuma szerint nem rendezett. Az index ritkaságából adódóan, minden adatblokkra csak egy mutató mutatna, így ezekből a rendezetlenség miatt nem tudnánk következtetni az adott blokk többi rekordjára.

Egy index létrehozásának SQL utasítása például az alábbi lehet:

CREATE INDEX szem_idx ON Személyek(szüldat);

A fenti utasítás a *Személyek* tábla *szüldat* attribútumára hoz létre egy *szem_idx* elnevezésű indexet.

A bemutatott néhány indexelési lehetőségből könnyen észrevehető, hogy az indexek hogyan gyorsítják fel a kereséseinket.

Egyik legfontosabb előnyük, hogy az indexblokkok száma jóval kevesebb lesz, mint az adatfájl blokkjainak száma, mert az indexben a teljes rekordsorok helyett csak kulcs és mutató párok vannak. Ritka index esetén ez még jobban tükröződik. Ha az index kellően kisméretű, akkor a leggyakrabban használt blokkjai, vagy akár az egész index, a központi memóriában is tárolható. A következmény az, hogy az I/O műveletek költsége jelentősen lecsökken, akár teljesen el is tűnhet, meggyorsítva ezzel a kereséseinket. A sűrű index legnagyobb előnye, hogy a „létezik-e *K* kulcsértékkel rendelkező rekord” típusú lekérdezésekre azonnali válasz adható beolvasás nélkül is. Ritka index estén már legalább egy I/O műveletre szükség van a kérdés megválaszolásához, ám így is kisebb a költségünk, mintha egyáltalán nem használnánk indexelést.

Az indexek másik fontos tulajdonsága, hogy kulcsértékeik szerint rendezettek, kereséskor bináris keresés alkalmazható. Ha n darab indexblokkunk van, akkor mindössze $\log_2 n$ blokkot kell csak átvizsgálnunk.

Hátrányuk viszont, hogy bármilyen adatkezelés esetén, ha törlésről, beszúrásról vagy felülírásról van szó, az adatmódosításokat minden alkalommal az indexben is el kell végezni. Ezek jelentős költséggel járhatnak.

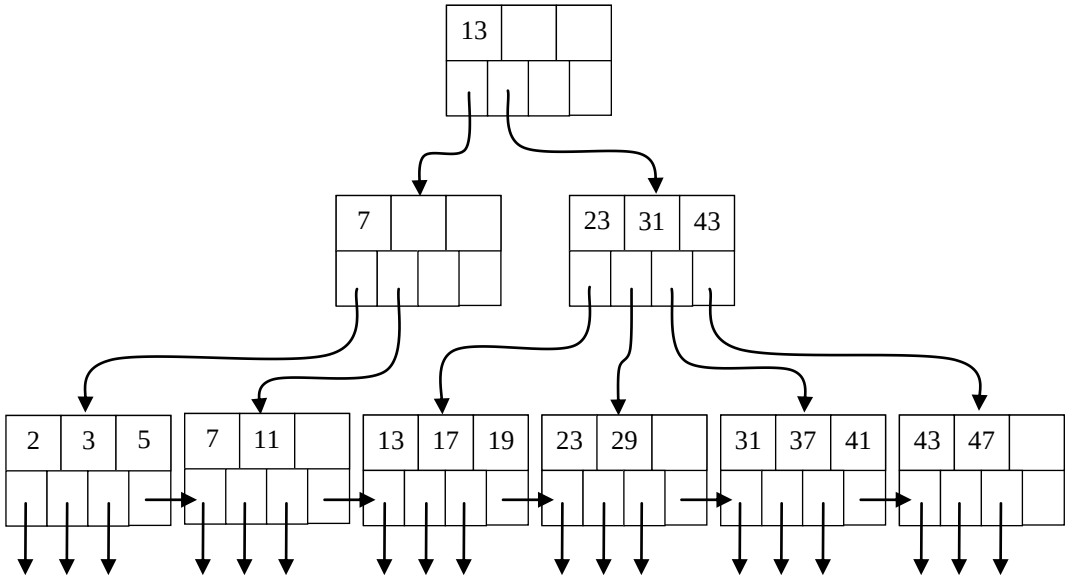
Érdemes a különböző lehetőségekből való választáskor ezeket a tényezőket figyelembe venni.

Abban az esetben, ha az index továbbra sem fér el a központi memóriában, újabb index készíthető fölé. Ennek menete pontosan megegyezik az eddig leírtakkal. Az indexet egy rendezett szekvenciális fájlnak tekintjük, és erre építünk további indexeket, még hatékonyabbá téve a kereséseket. A többszintű indexelés tulajdonsága, hogy az első szint fölé még bármennyi további szint készíthető, azonban minden további szint csak ritka lehet. Sűrű eset szóba sem jöhet, hiszen akkor nem lenne értelme a műveletnek, az indexrekordok száma nem csökkenne. Az így létrejött adatstruktúrát többszintű indexnek nevezzük.

2.3 B-fa index

Többszintű indexek alkalmazásakor már nagyobb problémát jelent, ha az eredeti adattáblában változtatásokat végzünk, hiszen ekkor az indexben is változtatásokra van szükség. Akár az is lehet, hogy az index minden szintjén módosításokat kell majd végrehajtani. Érdemes ezért egy olyan struktúrát alkalmazni, amely az adatok módosításakor könnyen kezelhető, illetve a keresést is a legjobban segíti.

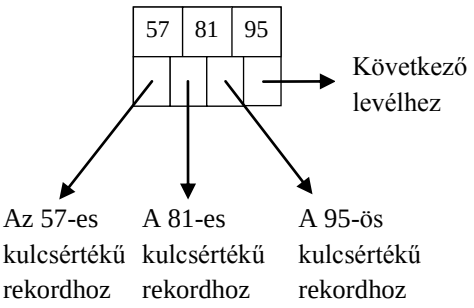
Egy ilyen elterjedt adatszerkezet, a korábbi tanulmányokból, az algoritmusok és adatstruktúrák területéről is jól ismert B-fa, illetve ennek a sok helyen használt változata a B^+-fa. Az Oracle is ezt a hatékony adatszerkezetet használja rekordjainak indexelésére. Egy B-fa a blokkjait fastruktúrába rendezi (2.5. ábra). Minden blokkja legalább félig telített. Egy ilyen fa kiegyensúlyozott, azaz minden, a gyökértől bármely levélig vezető út egyforma hosszú. A későbbiekben látni fogjuk, hogy bizonyos műveletek estén milyen plusz költségeket jelent egy fa kiegyensúlyozott tulajdonsága. A fához tartozik egy *n* paraméter, melynek jelentése az, hogy a fa egyes csúcsai *n* számú keresési kulcsot és *n* + 1 számú mutatót, azaz hivatkozást tárolnak. Tehát egy indexblokk *n* darab kulcs-mutató párból áll, és minden blokkhoz tartozik még egy pluszmutató is. Feltéve, hogy egy B-fát a szokásos módon ábrázolunk, azaz egy csúcs gyermekeit balról jobbra tűntetjük fel, akkor ezen irányban haladva a csúcsok kulcsai nem csökkenő sorrendben lesznek láthatók.



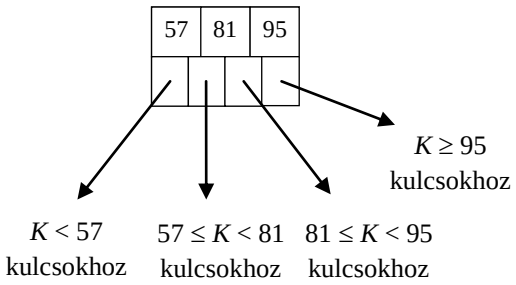
2.5 ábra: B^+-fa

A B-fa blokkjaira továbbá az alábbi megszorítások vonatkoznak [3]:

- **gyökérblokk esetén**: van legalább két használatban lévő mutató és minden mutató a következő szint egy blokkjára mutat.
- **levélblokk esetén**: n mutatóból legalább $\left\lfloor \frac{n+1}{2} \right\rfloor$ használatban van és adatrekordra mutat. Az i-edik mutató ($i \in [1..n]$), ha használatban van, akkor az i-edik kulccsal rendelkező rekordra mutat. Az $n+1$-edik mutató a következő levélblokkra mutat (2.6 ábra).
- **közbülső blokkok esetén**: mind az $n+1$ mutató a következő szint egy blokkjára mutat és közülük legalább $\left\lceil \frac{n+1}{2} \right\rceil$ használatban van. Tegyük fel, hogy i mutató van használatban, tehát i-1 kulcs van a blokkban ($K_1, K_2, \ldots, K_{i-1}$). Az első mutató a fa olyan részére mutat, amelyben a rekordok kulcsa kisebb, mint K_1. A második mutató a fa azon részére mutat, melyben a rekordok kulcsa nagyobb vagy egyenlő, mint K_1, de kisebb, mint K_2. Folytatva ezt az elméletet a B-fa egyes közbülső blokkjainak felépítése és egymáshoz való viszonyuk felrajzolhatóvá válik (2.7 ábra).



2.6 ábra: B⁺-fa jellegzetes levele



2.7 ábra: B⁺-fa jellegzetes belső csúcsa

A röviden ismertetett B-fák szerkezetéből észrevehető, hogy milyen lehetőségeket nyújtanak többszintű indexek készítéséhez. Jellemző tulajdonságuk ugyanis, hogy a levelekben található, adatrekordokhoz vezető mutatók sorozata, éppen megfelel a már korábban bemutatott indexek mutatósorozatainak. Azaz, egy levél megfeleltethető egy indexblokknak. Ezt figyelembe véve, részletezés nélkül megemlítem, hogy segítségükkel készíthető sűrű és ritka többszintű index is. Ha pedig a belső csúcsokra való megkötést a megfelelően módosítjuk, akkor támogatják a keresési kulcs ismétlődését is.

A B-fák legnagyobb előnye a hatékonyságukban rejlik. Lehetővé teszik rekordok gyors keresését rekurzív módon. Egy adatrekord keresésekor a fa gyökerétől kell eljutni a megfelelő levélig. A kereséshez felhasznált I/O műveletek száma meg fog egyezni a fa szintjeinek számával. Egy átlagos méretű adatbázishoz elég, hogy a fának három szintje legyen. A beolvasások száma tovább csökkenthető, ha a fa gyökerét vagy a második szinten lévő csúcsokat közvetlenül a központi memóriában tároljuk.

Erőteljesen támogatják a tartományra vonatkozó lekérdezéseket. Erre egy példa SQL lekérdezés:

SELECT TAJszám, név FROM Személyek WHERE életkor >= 25 and életkor <= 30;

A beszúrás (csúcsvágás lehetséges) és a törlés (csúcs összevonás előfordulhat) a jól ismert algoritmusok segítségével történik. Ezen két művelet esetén a beolvasásokon kívül felmerülnek adatmanipulációs költségek is. A fa újrarendezéséhez szükséges I/O műveletek száma minimálisra csökkenthető, ha az *n* paramétert elég nagynak (pl. 10-nél többnek) választjuk. Ugyanis, ilyen nagy *n* esetén csúcsvágásokra és csúcs összevonásokra egyre ritkábban lesz szükség vagy a műveletet kizárólag a levelekre kell elvégezni, azaz két levél és azok szülője lesz érintett.

Az indexek osztályozásának rövid leírásából látható, hogy mindig az adott keresési feltételeknek legmegfelelőbbet célszerű alkalmazni. A döntéskor figyelembe kell venni, hogy a tervezendő adatbázisunkon előreláthatólag milyen típusú lekérdezéseket fogunk előnyben részesíteni. A B-fákat optimális teljesítményük miatt már számos adatbázis-kezelő rendszerben használják. Ahogy a bevezető elején is említettem léteznek egyéb egydimenziós indexelési lehetőségek is, ilyenek például a hasítás módszerén alapuló tördelőtáblázatok.

3. Térbeli adatok tárolása és indexelése

Nagyméretű térinformatikai rendszerek (*GIS – Geographical Information System*) esetén speciális adatmodellezésre van szükség. A specialitást az adja, hogy a térképi rendszerek alapját képező adatbázis-kezelő rendszer alkalmas térbeli adatok tárolására is. Minden térbeli objektum két fő komponensből áll: egy térbeli komponensből, amely a grafikus megjelenítésért felelős, illetve egy leíró komponensből, amelyet táblázatos formában jelenítünk meg. Problémát okoz, hogy nehezen tartható fenn a grafikus és az attribútum adatok közötti konzisztencia. Az Oracle 11g Spatial kiterjesztése (térbeli modul) éppen ezeknek a problémáknak a megoldására jött létre, de ugyanezen célt szolgálja a PostgreSQL-nek a PostGis modulja is. A geometriai objektumok a relációs adatbázisban, mint absztrakt adattípusok kerülnek tárolásra, és a különböző típusú geometriai elemekhez önálló relációs tábla tartozik. Ez a felhasználó számára az egységes relációs megközelítés rugalmasságát és biztonságát adja.

A következő két fejezetben bemutatom azokat az adattárolási lehetőségeket, amelyek bármely, téradatokat is kezelő, adatbázis-rendszer esetén alkalmazhatók. Dolgozatom további, bármely fejezetében megfogalmazott módszerek az egyszerűség kedvéért a 2D térre, azaz a síkra értendők, de általánosíthatók magasabb dimenziókra is.

3.1 A térinformatikai adatmodellek

A térbeli (most 2D) objektumokat általában két csoportra lehet osztani: a raszteres adatmodellre épülők, és a vektoros adatmodellre épülők, úgynevezett vektoros térképek. Mindkét modellt csak érintőlegesen mutatom be, két kisebb alfejezetben.

3.1.1 A raszteres adatmodell

A raszteres modell adatforrása egy digitális kép, amely általában egy műholdfelvétel, de lehet szkennelt kép, illetve digitális fényképezőgéppel készült felvétel is.

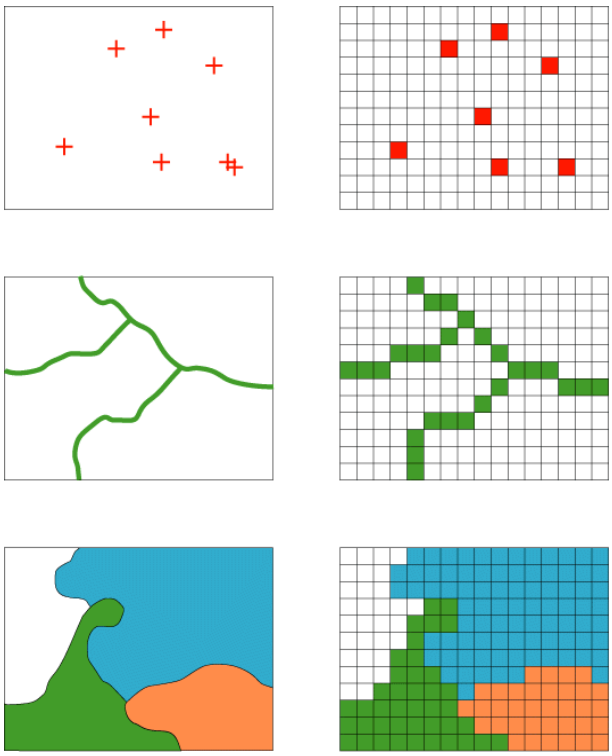
A digitálisan készült felvétel a földfelszín szabályos négyzet rácshoz rendelt számokból alkotott sorozata, azaz egy képmátrix, amelynek az elemei a pixelek. A mátrix elemei azonos dimenziójú egész típusú vektorok. A közös dimenzió a felvétel készítésénél használt sávok száma, a vektorok értéke az egyes sávokban mért intenzitás értékek. Egy pixel a modell elemi objektumát képezi, amely egy adott területegységet fed le és a felbontás adja meg, hogy az általa lefedett terület mérete mekkora.

A képalkotó rendszerek tehát a megfigyelt területegységet pixelekre, az egész területet egy $n \times m$-es mátrixra képezik le (n a sorok, m az oszlopok száma) és a mátrix celláiban lévő képpontok állapota rajzol ki egy valamilyen felismerhető objektumot. Ez a raszteres reprezentáció legegyszerűbben kezelhető esete. Létezik olyan módszer, amelyben a teret nem szabályos négyzetekre, hanem egyéb sokszögekre bontjuk fel.

A raszteres adatmodell a pixeleken kívül olyan lényeges információkat is tartalmaz, amelyek megadják a rasztert felépítő pixelek sorainak és oszlopainak számát és a georeferencia adatokat (általában a bal felső pixel középpontjának koordinátáit és a pixel által lefedett terület méreteit). Ezek együttesen adják a földrajzi objektumok geometriai jellemzését. A raszteres adatbázis raszterekből áll és az egyes raszterek képviselik az objektum osztályokat.

A modell jellemző tulajdonságai [5]:

- minden pixel állapotát ismerni kell. Speciálisan az üres pixelek értéke 0. Ennek eredményeképpen nagyméretű adatbázisokra van szükség.
- nehezen definiálhatók az objektumok
- nehezen kiépíthető relációs adatbázis-kapcsolatok, de könnyen vizsgálható geometriai relációk
- gyors, egyenletes mintavételezés



3.1 ábra: Pontok, vonalak és poligonok raszteres rendszerben

3.1.2 A vektoros adatmodell

A vektoros modellben az objektumok leírása helyvektorok segítségével történik. Az objektumok térbeli elhelyezkedését megadó jellegzetes pontokat helyvektoruk koordinátájával tároljuk. Egy adott területen lévő objektumok viszonylag kevés ponttal jellemezhetők. A koordináták mellett egy összekötési szabályt adunk meg arra, hogy a pontok milyen sorrendben legyenek összekötve. A korrekt leíráshoz objektumdefiníciók is szükségesek, amelyek megmondják, hogy mely pontok alkotnak egy alakzatot.

Egy vektoros rendszerek önálló geometriai elemei:

- **pontszerű (0D) objektum (point)**: az objektumnak a koordinátáit tároljuk az adatbázisban. Definíció: *point:* [*x*: *real*, *y*: *real*]

  Két típusa van:

  - *kis méretű objektum*: az adott méretarány mellett túl kicsi a grafikus ábrázoláshoz (pl. lámpaoszlop). Megjelenítése meghatározott jelkulcsi jelöléssel történik.
  - *csomópont*: vonalas objektumok találkozási helyét jelöli (pl. útelágazás). Rendszerint nincs külön grafikus megjelenítése.

- **vonalas (1D) objektum (polyline)**: a vonalas objektum egy vonallánc, amelyet pontok sorozatával adunk meg. A töréspontok koordinátáit tároljuk az adatbázsiban. (Pl. folyó) Definíció: *polyline:* < *point* >

- **területi (2D) objektum (polygon és region)**: egy poligont szintén pontok sorozatával adunk meg. Abban különbözik a vonalas objektumtól, hogy a poligon egy zárt vonallánc. A jellegzetes csúcspontok koordinátáit tároljuk az adatbázisban. (Pl.: telek) Definíció: *polygon:* < *point* >

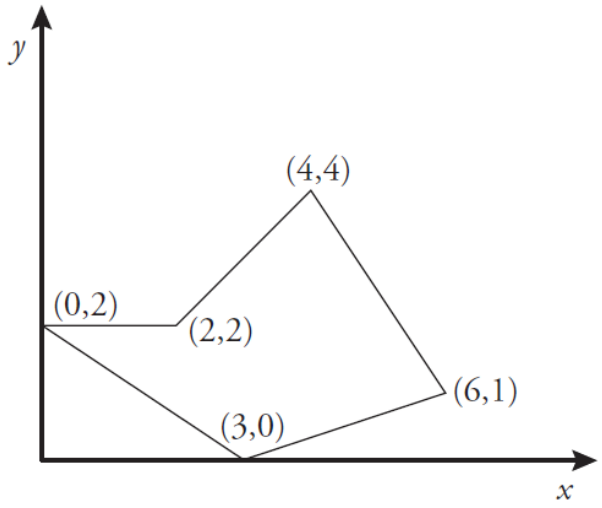
  A területi objektumok poligonok halmaza. Megjelenítésük adott kitöltő mintázattal és/vagy színnel történik. Definíció: *region:* { *polygon* }

Adatbázis-kezelő rendszertől függően lehetnek további beépített elemek is. Pl. az Oracle adattípusa, az **SDO_GEOMETRY**, tartalmazza az előbbieken felül az íveket és a köröket (ellipsziseket) is.

A vektoros rendszerek legnagyobb előnye, hogy az elemi objektumokból, azok különböző kombinációit véve, egyre összetettebb objektumok építhetők fel. A modell elemi építőkövei a node-ok, melyekből egy szabályrendszer segítségével tudunk komplex objektumokat létrehozni. A komplex objektumokhoz pedig olyan mezők definiálhatók, melyek intelligens kapcsolatok felépítését tetszik lehetővé.

A modell jellemző tulajdonságai [5]:

- alkalmas objektumorientált szemléletű rendszerek készítésére
- kevés adattal, kisméretű adatbázissal is képes a felület megfelelően pontos leírására
- alkalmas relációs adatok hozzákapcsolására
- gyors, egyenletes mintavételezés



3.2 ábra: Egy poligon vektoros ábrázolása

Napjainkban a térinformatikai rendszerek többsége a vektoros adatmodellt használja, mert alapvető fontosságú a leíró jellegű adatok hozzákapcsolása a térképekhez. A raszteres és a vektoros adatok együttes megjelenítése hasznos funkció lehet, mert mindkét adatmodell előnyei kihasználhatóak és a két modell egymást jól kiegészíti. (megjelenítés és adatkezelés)

3.2 Térbeli indexelés

Az 1970-es években, a számítógépes geometriának egy virágzó területe volt a központi memória részére 2D-s térbeli struktúrák tervezése. Később, a ’70-es évek végén, a külső memóriabeli szerkezetekre való igény az alkalmazásokat, így a GIS-t is, a további kutatásokra ösztönözték, amelyek erősen érintették a térinformatikai adatbázisokat is.

Az első fejezetben bemutatott indexstruktúrák az egydimenziós, relációs adatbázisban tárolt adatokra hatásosan alkalmazhatók. Ahogy a térinformatikai rendszerek adatmodellezése speciális módon történik, úgy az indexelés területén is a

hagyományostól eltérő módszerekre van szükség. Az egydimenziós indexstruktúrák nem támogatják a térbeli alakzatokra vonatkozó lekérdezéseket.

A legnagyobb probléma, hogy a többdimenziós teret nem lehet távolságtartással leképezni egydimenziósra, hogy a hagyományos indexek alkalmazhatók legyenek.

További nehézséget jelent az egydimenziós adatoknál is említett tárigény és az adatmozgatások a háttértároló és a központi memória között. A digitális térképek kezelése – raszteres, és vektoros térképek esetén is – nagy erőforrás igényű, sok memóriát igényel és nagy I/O költségeket okoz, ha nem megfelelő módszert választunk erre a célra. Egy-egy lekérdezés eredményének meghatározásához az összes objektum végignézése elfogadhatatlanul lassú lenne. Ha egy térképet szeretnénk megjeleníteni, akkor annak általában csak egy ablakába (*viewport*) eső területre van szükségünk, így felesleges költséget jelent minden elem beolvasása, majd kirajzolása. A gyors keresésekhez és leválogatásokhoz van szükség a térbeli indexelési algoritmusok ismeretére és alkalmazására.

Az algoritmusok áttekintéséhez szükséges a legkisebb befoglaló téglalap fogalmának ismerete, amelynek segítségével könnyen reprezentálhatók a rendkívül összetett objektumok is. Objektumok reprezentálására használt másik jellemző adat az alakzatok középpontja (*centorid*). A továbbiakban feltettem azt is, hogy minden objektumnak egyedi azonosítója van, így az indexelési módszerek végrehajtásához elegendő a legkisebb befoglaló téglalap (*mbb*) és az objektumazonosító (*oid*) ismerete. A lekérdezések eredményének kiválasztása két lépésből áll. Egy előszűrésből, amikor kiválogatjuk azokat az *oid*-ket, amelyek szóba jöhetnek az *mbb*-jük alapján. A második lépés maga a kiválasztás (finomítás).

A térbeli indexelési módszerek osztályozása a paraméterek nagyszámú fejlődése miatt nagyon nehéz. Az általam bemutatásra kerülő két osztály (területalapú és adatvezérelt) túl egyszerű ahhoz, hogy figyelembe vegye az indexstruktúrák nagyon sokféle variációját. Mivel nem az összes indexelési lehetőség ismertetése volt a célom, ezért az áttekinthetőség és érthetőség miatt én mégis erre a két egyszerű csoportra bontottam őket.

Bizonyos tekintetben még egy osztályozási szempontot lehetne bevezetni a következő fejezetekben szereplő indexekre, mégpedig, hogy milyen jellegű objektumokat indexelünk velük: pontszerű (*centroid*) vagy terület alapú (*mbb*). A pont alapú módszereknél, mivel pontokat indexelünk, nincs szükség a szűrő és finomító eljárásokra. Ezzel szemben a területalapú módszerek vonalakkal és poligonokkal

dolgoznak, oly módon, hogy az objektumokat azok befoglaló téglalapjával azonosítjuk (közelítjük). Ez esetben szükséges a szűrő lépés, amely az *mbb*-vel dolgozik és a finomító eljárásra, amely már a konkrét objektumokat kezeli. Ezen két osztály között egyszerűen találhatunk egy megfeleltetést, mert minden téglalap leképezhető a 4D-s pontok halmazára úgy, hogy egy téglalapot a négy sarokpontjával azonosítunk. Ezután a pont hozzáférésű indexelési technikák bármelyike alkalmazható lesz téglalapokra is.

Bármely szakirodalmat, összehasonlító tanulmányt vagy könyvet elolvasva, nem találunk egységes indexelési módszert, amely hatékonyan működne a különböző térbeli lekérdezések esetén. Ilyen tipikus térbeli lekérdezések pl. a következők lehetnek:

- adott pont elhelyezkedése és pozíciója (pontkeresés).
- adott terület lekérdezése, azaz a területbe eső objektumok vagy azok pozíciójának meghatározása.
- távolságokra vonatkozó lekérdezések (két pont távolsága, két poligon távolsága, pont és vonal távolsága stb.).
- a távolságokhoz köthető további lekérdezések, mint pl. (*k*) legközelebbi szomszéd problémája és a korlátos (*k*) legközelebbi szomszéd problémája, ahol a korlát egy minimum vagy egy maximum távolságot ad meg.
- térbeli összekapcsolással kapcsolatos lekérdezések: metszetek, átfedések, érintkezések, kereszteződések és tartalmazások.

A következő két fejezetben, ahogy azt az előzőekben említettem a térbeli indexek két csoportjára helyeztem a hangsúlyt. A struktúrák valamely osztályba való besorolása jelen esetben attól függ, hogy a teret osszák fel, vagy a térben elhelyezkedő objektumokból indulnak ki és közelítik azokat. Az előbbieket területalapú módszereknek, az utóbbiakat adatvezérelt módszereknek nevezzük.

4. Területalapú módszerek

A területalapú módszerek közös jellemzője, hogy az alakzatok helyett a térből indulnak ki és azt bontják fel az adott algoritmusnak megfelelő alakú és méretű cellákra és így végzik a műveleteiket.

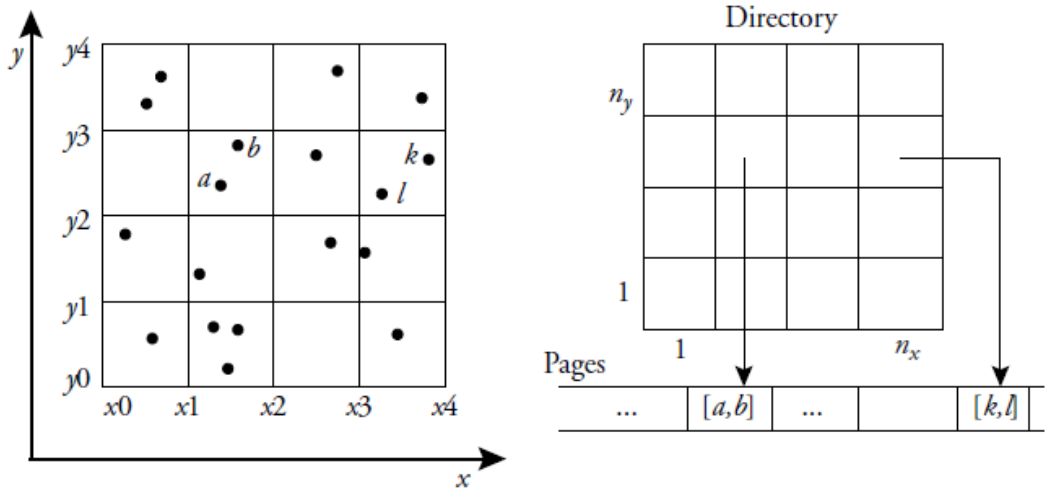
4.1 Grid index

A grid index a legegyszerűbb térfelosztó indexelési eljárás, amelynek a teljesítményét tekintve gyakran sokkal hatékonyabb az egydimenziós indexelési eljárásoknál.

4.1.1 Fix grid

A tárolt térrész mérete az *x* tengely irányában S_x, az *y* tengely irányában S_y. A keresési tér négyzet alapú cellákra van felosztva, azaz egy $n_x \times n_y$ mátrix reprezentálja. A cellák mérete S_x / n_x és S_y / n_y lesz. A mátrix struktúra miatt nevezik a módszert rácsos állománynak.

Pontok indexelése [1]: A tér felosztásával létrejött mátrix minden cellájához hozzá van rendelve a háttértároló egy blokkja és ezekre a blokkokra mutató pointereket egy $DIR\,[1:n_x, 1:n_y]$ kétdimenziós vektorban tároljuk (4.1 ábra). A $DIR\,[i, j]$ egy eleme tartalmazza annak a lapnak a címét (*PageID*), amely azon pontokat tárolja, amelyek a c_{ij} cellához tartoznak, azaz amelyeket a c_{ij} cella tartalmazza.

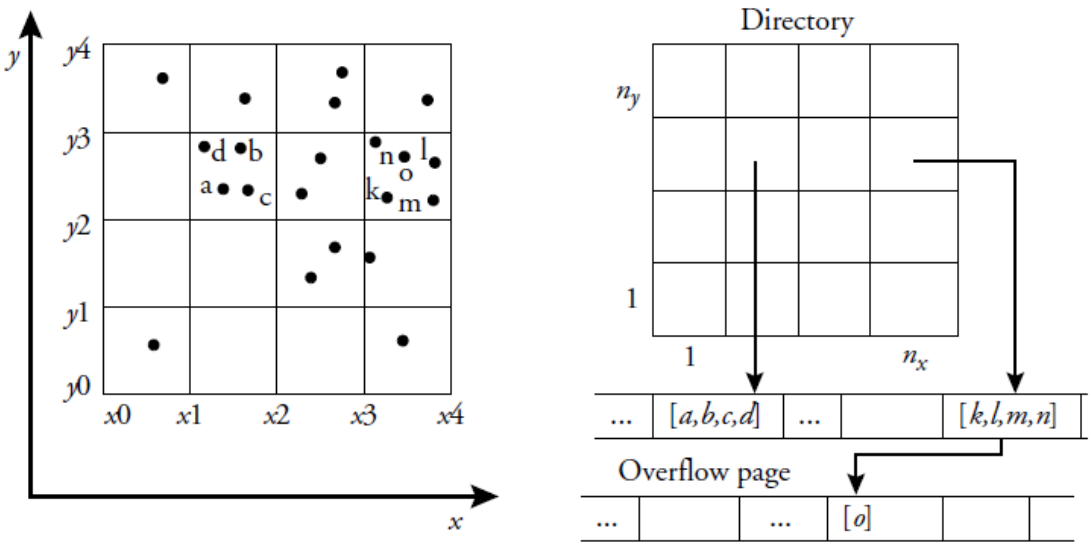


4.1 ábra: Pontok indexelése fix grid segítségével

Az előző jelöléseket alkalmazva a pont és a terület lekérdezés algoritmusai a következőképpen írhatók fel:

- **Pont lekérdezés**: A P pont a és b koordinátája alapján kiszámítható i és j a következő módon: $i = (a - x_0) / (S_x / n_x) + 1$, $j = (b - y_0) / (S_y / n_y) + 1$. Ezek segítségével a referenciamátrixban megtalálható a P pontot tartalmazó c_{ij} cellához rendelt $DIR\,[i, j].PageID$. A lapot beolvassuk és megnézzük, hogy szerepel-e benne az adott pont. Beszúráskor és törléskor ugyanezt az eljárást alkalmazhatjuk, majd a beolvasás után végrehajtjuk a beszúrást vagy a törlést.
- **Terület / ablak lekérdezés**: Ki kell számítani azt az S halmazt, amely tartalmazza a lekérdezés paramétereként szereplő W ablakot lefedő c cellákat. Minden ilyen c_{ij} cellára beolvassuk a megfelelő $DIR\,[i, j].PageID$ lapot, és azokban megkeressük a W ablak összes pontját.

Ha a referenciamátrix elfér a központi memóriában, akkor a pont lekérdezésekhez egyetlen I/O műveletet kell csak végrehajtani. Ellenkező esetben a mátrix mérete lesz a meghatározó. Terület lekérdezések esetén az I/O műveletek számát az ablak által lefedett cellák száma határozza meg. A grid felbontása az indexelendő pontok és objektumok számától illetve a lapmérettől függ. Ha a pontok száma N és a memória kapacitása M, akkor legalább N / M db cellával kell lefedni a teret. Ha egy cella M-nél több pontot tartalmaz, akkor túlcsorduló celláknak nevezzük és szükség van további, úgynevezett túlcsordulási lapokra. Ezek a túlcsordulási lapok az eredeti lapokhoz rendelődnek hozzá.



4.2 ábra: Túlcsordulási lapok használata

A 4.2 ábrát tekintve az látható, hogy a lapok átlagosan 4 pontot tudnak tárolni. A *k*, *l*, *m*, *n* és *o* pontokat tartalmazó cella esetén a *k*, *l*, *m* és *n* pontok a normál lapon tárolódnak, míg az *o* pont a túlcsordulási lapra kerül át, és az eredeti lap ennek a túlcsordulási lapnak a memória címét fogja tartalmazni. Ha a pontok eloszlása közel egyenletes a térben, akkor ritkán van szükség túlcsordulási lapokra. Legrosszabb esetben az összes pont egy cellába esik, ekkor az adatszerkezet túlcsordulási lapok sorozata lesz. Ebben a struktúrában való keresés időben lineáris, azaz nem túl hatékony.

Téglalapok indexelése [1]: Most megvizsgálom, hogy hogyan lehet összetett objektumokat indexelni. Elegendő a módszert téglalapokra alkalmazni, mert minden alakzatnak van minimális befoglaló téglalapja, így az összetett objektumok indexelése visszavezethető a téglalapok indexelésére.

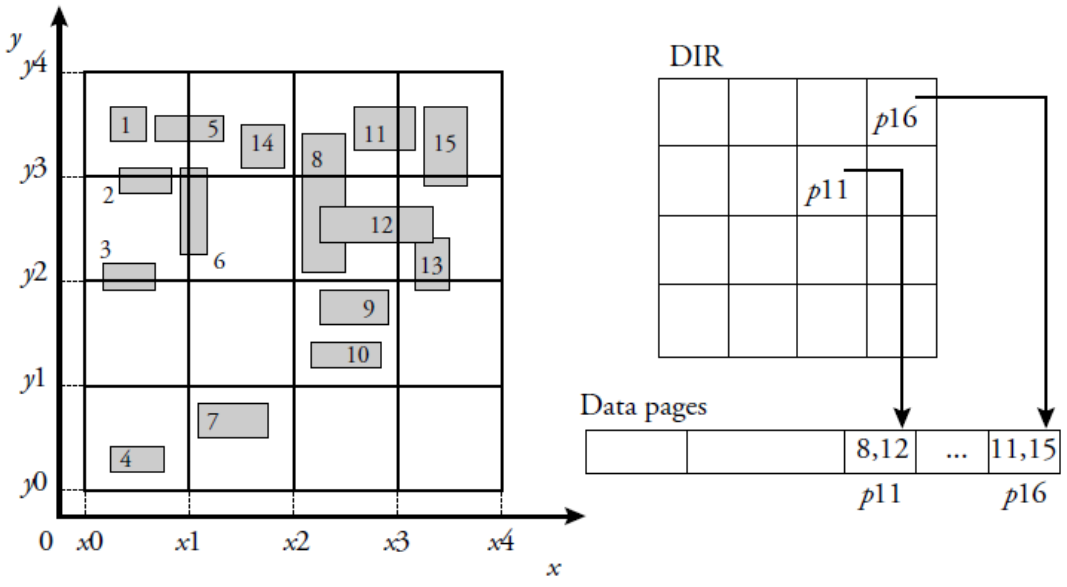
A pontok indexelésénél bevezetett jelöléseket alkalmazva megmutatom, hogyan lehet téglalapokat indexelni. Azt mondjuk, hogy egy *c* cellához tartozik egy *mbb*, ha az alábbi három eset közül az egyik teljesül:

(1) a téglalap tartalmazza a cellát

(2) a cella tartalmazza a téglalapot

(3) a cella és a téglalap metszik egymást

Ekkor az *mbb* minden, vele metszésben lévő cellához rendelt lapon meg fog jelenni az azonosítójával (4.3 ábra). Új téglalap beszúrása és törlése ugyanúgy történik, mint a pontok esetén, azzal a kivétellel, hogy (1) és (2) esetben lehetséges, hogy a végrehajtott művelet több lapot is érinteni fog, ezzel a költségeket növelve.



4.3 ábra: Téglalapok indexelése fix grid segítségével

Nagyméretű téglalapok indexelésekor a túlcsordulás valószínűsége nagyobb, ez az index méretének növekedésével és a hatékonyság csökkenésével jár.

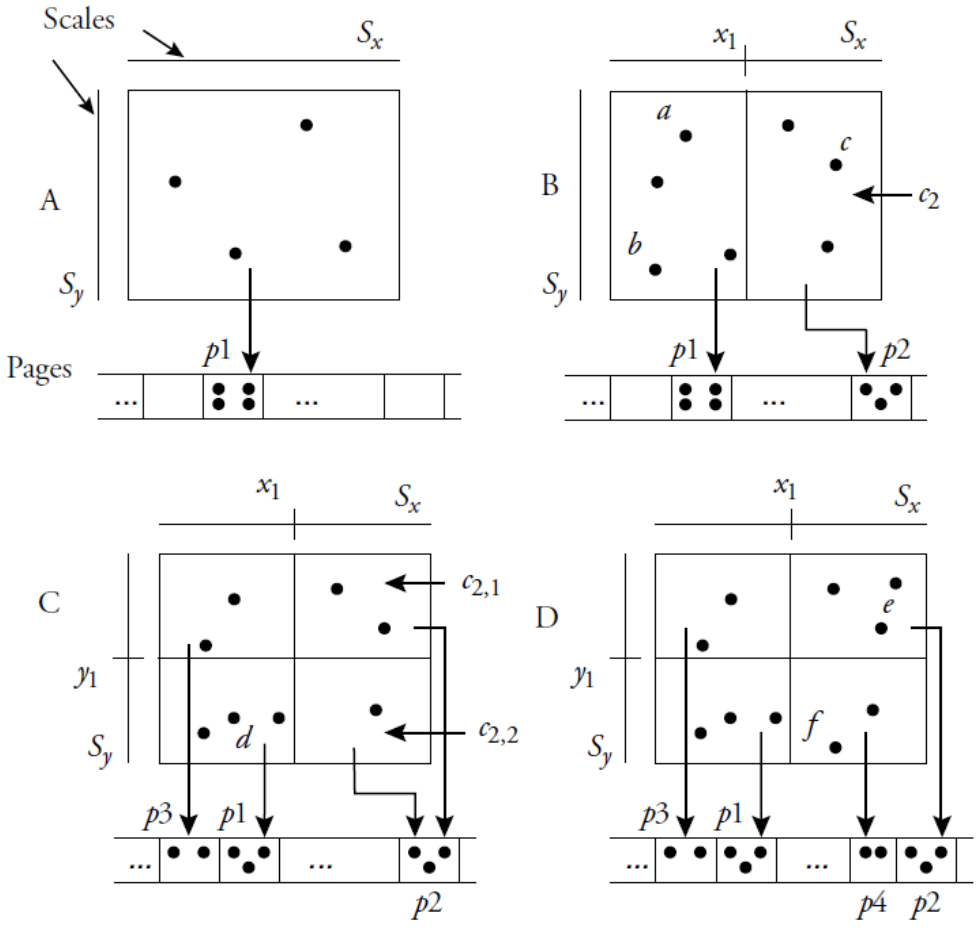
4.1.2 Grid file

A módszer abban különbözik a fix gridtől, hogy kezdetben nincs térfelosztás.

Pontok indexelése [1]: Új pont beszúrása után szükséges megvizsgálni, hogy nem történt-e túlcsordulás a blokkban. Ha túlcsordulás volt, akkor cellafelezést alkalmazunk és minden pont az őt tartalmazó megfelelő cellába kerül. A mátrix rácsaihoz továbbra is a háttértároló blokkjai vannak hozzárendelve, de egy blokkhoz több cella is tartozhat.

Az adattároláshoz szükséges három adatszerkezet:

- *DIR* elnevezésű, blokkcímeket tartalmazó mátrix. A fix grid mátrixától az különbözteti meg, hogy egy blokkra több szomszédos cella is hivatkozhat.
- Két lista S_x és S_y, amik a tengelyek intervallumokra bontását tartalmazzák. Minden cellafelezéskor belekerül az S_x vagy S_y mentén keletkezett új osztópont.



4.4 ábra: Pontok indexelése grid file segítségével

A 4.4 ábrát tekintve az látható, hogy a lapok átlagosan 4 pontot tudnak tárolni. Az **A** esetben mind a 4 pont elfér egy blokkban, nincs túlcsordulás így nincs szükség cellavágásra sem. A **B** esetben beszúrjuk az *a*, *b* és *c* pontokat. A p_1 blokk túlcsordul, ezért S_x mentén felveszünk egy x_1 felezőpontot, függőlegesen kettévágjuk a cellát, majd egy p_2 blokkot allokálunk és minden pontot a megfelelő blokkba teszünk. A **C** esetben, a *d* pont beszúrásakor a p_1 blokk ismét túlcsordul, ezért most S_y mentén felveszünk egy y_1 felezőpontot, vízszintesen kettévágjuk a cellát, majd egy p_3 blokkot is allokálunk. Megfigyelhető, hogy a p_2 blokkhoz tartozó cella is két részre szakadt, de mindkettő továbbra is ugyanarra a blokkra mutat. Ez azért van, mert a *d* pont beszúrásakor a vágás megtörtént, de a p_2 blokkban nem volt túlcsordulás. Két újabb pont, *e* és *f* beszúrásakor már p_2 is túlcsordul, de cellavágásra már nincs szükség, ugyanakkor allokálni kell egy új p_4 blokkot. Ezt illusztrálja a **D** eset.

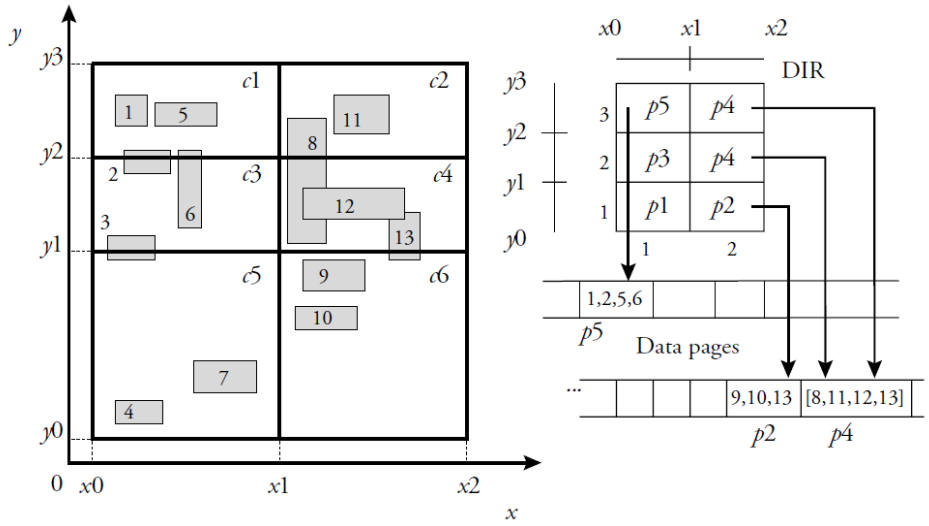
A példából látszik, hogy egy pont beszúrásakor négy esetet különböztetünk meg:

- **nincs cellavágás:** Ez a legegyszerűbb eset, amikor a beszúrt pont nem okoz blokk túlcsordulást, ezért nincs szükség újabb blokk allokálására (4.4 ábra / A).
- **van cellavágás, de nem allokálódik új blokk:** Egy cellavágást mindig egy blokk túlcsordulása idéz elő, de a vágás több cellát is érinthet, attól függően, hogy korábban, az adott tengely mentén voltak-e már osztópontok felvéve. Ha voltak, akkor az adott irányban egy újabb vágás biztosan több cellát is érinteni fog, de csak az egyikhez fog új blokk létrejönni. Ilyen esetekben fordul elő, hogy egy blokk több rácshoz is hozzá van rendelve (4.4 ábra / C).
- **nincs cellavágás, de allokálódik új blokk:** Ez az eset az előző következménye. Akkor fordul elő, ha egy blokk egyszerre több cellához is hozzá van rendelve és újabb pont beszúrásakor a cellákban szereplő pontok összessége már túlcsordulást okoz. Ilyenkor a rácsok már megvannak, csak új blokkot kell allokálni (4.4 ábra / D).
- **van cellavágás és új blokk allokálódik:** Minden cellavágáskor előfordul, de függetlenül attól, hogy az osztás hány cellát érint, mindig csak az egyik rács idézi elő új blokk létrehozását (4.4 ábra / B vagy 4.4. ábra / C bal oldali cella kettéosztása).

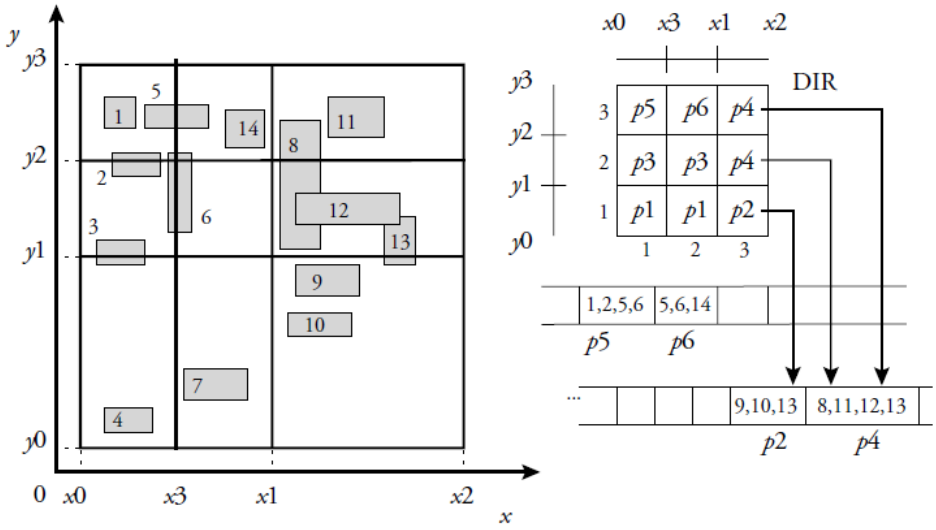
Műveletigény szempontjából sokkal előnyösebb a fix gridnél, mert nem használ túlcsordulási blokkokat. Ha a referenciamátrix teljes egészében elfér a központi memóriában, akkor pontkeresésnél, még ha túlcsordulás elő is fordul, elegendő egy I/O

művelet végrehajtása. Hátránya, hogy túl sok cella esetén a referenciamátrix biztosan nem fér el a központi memóriában.

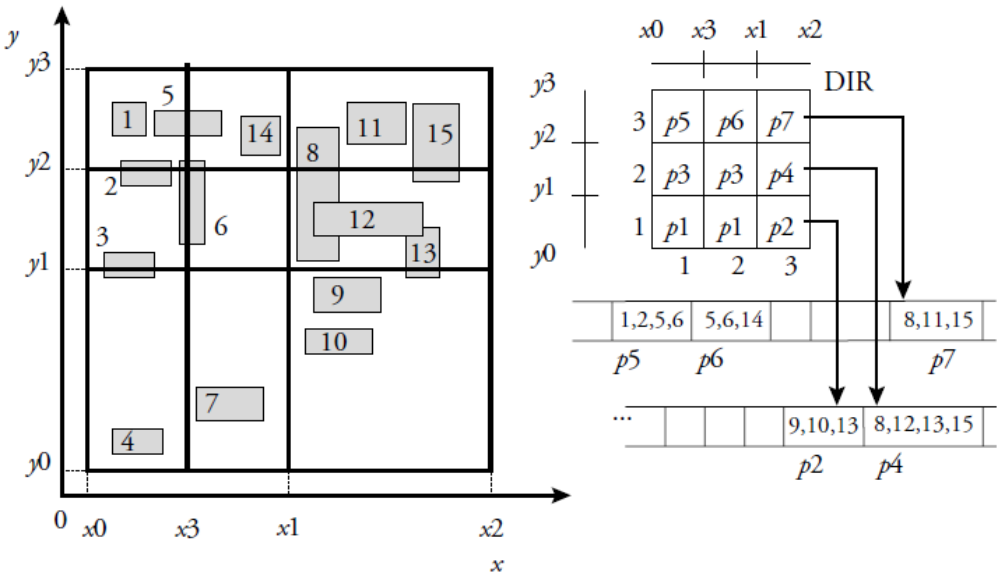
Téglalapok indexelése [1]: A fix gridhez hasonló módon történik a téglalapok tárolása a referenciamátrix celláihoz rendelt blokkokban. A különbség továbbra is a túlcsordulásoknál jelentkezik cellavágások formájában. A 4.5 eredeti állapotot tükröző ábrából kiindulva a 4.6 ábra mutatja a cellavágást és új blokk allokálását igénylő 14-es azonosítóval rendelkező téglalap beszúrása. A 4.7-es ábrán egy olyan eset látható, amikor egy új ($oid = 15$) téglalap beszúrásakor cellavágásra nem, csak új blokk allokálására van szükség.



4.5 ábra: Téglalapok indexelése grid file segítségével



4.6 ábra: 14-es téglalap beszúrása



4.7 ábra: 15-ös téglalap beszúrása

Téglalapok indexelésekor is felírhatók a pont és a terület lekérdezés algoritmusai:

- **Pont lekérdezés:** A lekérdezés lépései a következők lesznek:
  (1) A *P* pont koordinátái alapján meghatározzuk azt a cellát, amely a keresendő pontot tartalmazza. Az S_x és az S_y listákon bináris keresés alkalmazásával a cellakeresés időben logaritmikus lesz.
  (2) A pontot tartalmazó cella által hivatkozott blokkot beolvassuk a központi memóriába. Ennek műveletigénye egy lemezolvasás lesz.
  (3) A beolvasott blokkban tárolt *oid*-k közül kikeressük azokat, amelyekhez tartozó *mbb* tartalmazza a *P* pontot.

  A fenti jelölések bevezetése után az algoritmus a következő [1]:

GF-POINTQUERY(*P(a, b)* : Point) : Set(*oid*)

Begin
  res := 0
  i := RANK(a, S_x); *j* := RANK(b, S_y)
  page := READPAGE(*DIR* [*i*, *j*].*PageID*)
  for each *e* **in** *page* **do**
    if(*P* in *e.mbb*) **then** *res* += {*e.oid*} **end if**
  end for
  return *res*
End

Az algoritmus tehát azon objektumok azonosítóit adja vissza, amelyek legkisebb befoglaló téglalapja tartalmazza a *P* pontot. A *RANK*(*v*, *S*) függvény meghatározza, hogy az adott pont koordinátái mely skálába esnek az egyes tengelyeken. A *READPAGE*(*p*) beolvassa a *p* blokk tartalmát a memóriába. Mivel a komplex objektumoknak csak a legkisebb befoglaló téglalapjával dolgoztunk, egy további finomításra is szükség van, amely megmondja, hogy maga az objektum is valóban tartalmazza-e a keresett pontot.

- **Terület / ablak lekérdezés:** A lekérdezés lépései a következők lesznek:
  (1) Ki kell számítani azt az *S* halmazt, amely tartalmazza a lekérdezés paramétereként szereplő *W* ablakot lefedő *c* cellákat.
  (2) A pont lekérdezéshez hasonló algoritmust futtatunk az *S* halmazra és minden *c* cellát megvizsgálunk.

  Az eredmény többszörösen tartalmazhatja ugyanazokat az elemeket, ezeket rendezzük, majd a duplikátumokat töröljük.

  Az algoritmus a következő [1]:

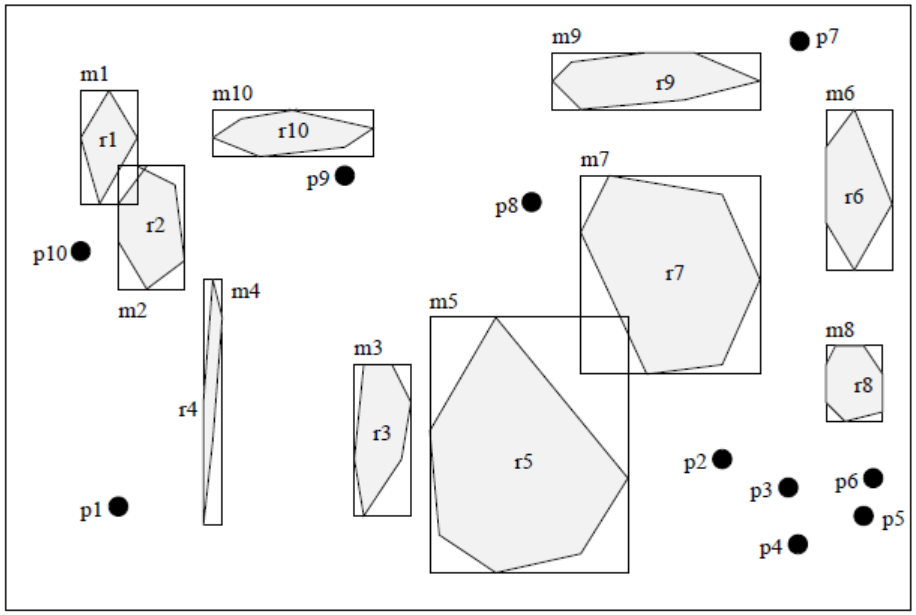
GF-WINDOWQUERY(*W*(x_1, y_1, x_2, y_2) : Rectangle) : Set(*oid*)

```
Begin
    res := 0
    i1 := RANK(x1, Sx); j := RANK(y1, Sy)
    i2 := RANK(x2, Sx); j := RANK(y2, Sy)
    for(i = i1; i <= i2; i++) do
        for(j = j1; j <= j2; j++) do
            page := READPAGE(DIR [i, j].PageID)
            for each e in page do
                if(W ∩ e.mbb ≠ Ø) then res += {e.oid} end if
            end for
        end for
    end for
SORT(res); REMOVEDUPL(res)
Return res
End
```

A költségeket tekintve több megállapítást is lehet tenni. A módszer hátránya, hogy a duplikálódott elemek megnövelik az index-bejegyzések számát, így az index méretét. Ez a hátrány az adatmennyiség növekedésével egyre jelentősebbé válik. A többszörös objektumok rendezése és törlése költséges és magas tárigényű. A hatékonyság növelhető, az egymást követő lapok betöltésének gyorsításával. Ha a blokkok közvetlenül egymás után következnek a lemezen, gyorsabban betölthetők a központi memóriába. (*n* darab, egymás után következő lap kiválasztása sokkal gyorsabb, mint *n* darab, véletlenszerűen elhelyezkedő lap kiválasztása.)

4.2 kd-fa

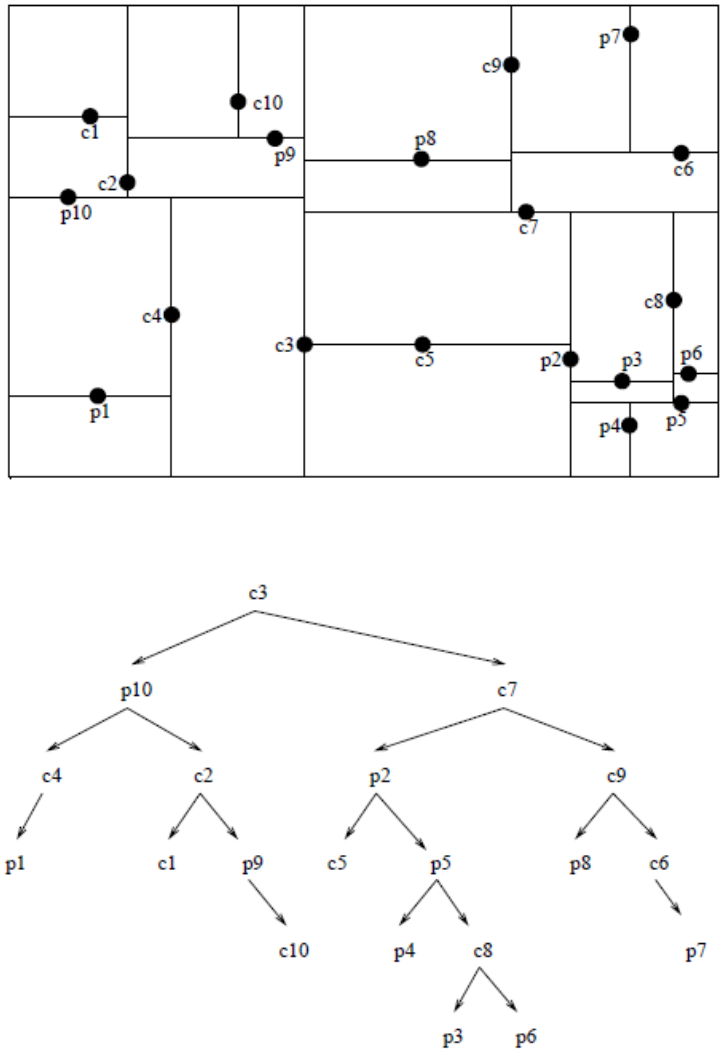
A kd-fa (*k-dimensional tree*) a legismertebb központi memóriabeli, többdimenziós adattárolási eszköz. Ez egy olyan bináris keresőfa, amely a tér kisebb terekre való felosztását reprezentálja. A térfelosztás egy rekurzív algoritmussal történik. A kd-fa pontokat kezel a csúcsaiban, ezért ebben az esetben az alakzatok legkisebb befoglaló téglalapja helyett, a középpontjukat (*centroid*) használjuk azonosításukra (4.8 ábra).



4.8 ábra: térbeli objektumok és azonosításuk

A rekurzív algoritmus egyes lépéseiben a *d* dimenziós tér valamely *d* - 1 dimenziós alterével (sík esetén koordináta tengelyekkel) párhuzamos hipersíkkal osztjuk ketté a teret úgy, hogy minden hipersík metszésben legyen legalább egy pontszerű objektummal, vagy *centroid*-dal a térből, amely a vágósíkot fogja reprezentálni a fában.

A fa újonnan beszúrt csúcsában ezt a pontot tároljuk el és minden ilyen belső csúcsnak egy vagy két gyermeke van. Tartozik a csúcsokhoz egy döntési algoritmus is, amely a fában való keresések irányát határozza meg. A vágósík két oldalán optimális esetben nagyjából azonos számú objektum van, amelyek a beszúrt csúcs bal oldali illetve jobb oldali részfájába kerülnek. Ennek következménye, hogy síkbeli kereséskor a fa páros szintjein az *x* koordinátákat, páratlan szintjein az *y* koordinátákat hasonlítjuk össze. Pl.: A fa egyik páratlan szintjén adott egy *P* csúcs. Minden olyan pont, amelynek az *y* koordináta-értéke kisebb, mint a *P* csúcs *y* koordinátájának értéke, a *P* baloldali gyerekének leszármazottjai között fog szerepelni. Ha ez az *y* koordináta-érték nagyobb, akkor a jobboldali részfában lesz megtalálható [6]. A vágások után létrejött 2 térrészre addig folytatódnak a rekurzív térfelosztások, amíg a térrészekben van még a síkok által nem metszett pont (4.9 ábra).



4.9 ábra: kd-fa

A 4.9 ábra a 4.8 ábrán látható 2D alakzatokból épített kd-fát ábrázolja. Először az r_3 objektumot a középpontjával (c_3) azonosítjuk, majd vesszük a c_3-at metsző és az y tengellyel párhuzamos egyenest és a síkot két részre osztjuk. A fa gyökerébe bekerül a c_3. Ugyanezt alkalmazzuk a keletkezett két térrészre is, és így tovább. Ezzel a módszerrel építjük fel a kd-fát. Minden esetben olyan objektumot választunk, amelynek két oldalán közel azonos számú alakzat helyezkedik el.

Egy adott pont keresése (pont lekérdezéskor) a gyökérből kiindulva történik. Az egyes szinteken a megfelelő koordinátákat összehasonlítva, addig haladunk lefele a fában, amíg el nem érjük a keresett pontot tartalmazó csúcsot.

A fa építésénél bemutatott algoritmussal tudunk egy új pontot beszúrni. A beszúrás végrehajtása során a csúcskeresés algoritmusát alkalmazzuk, azzal a különbséggel, hogy a keresés végén a megfelelő levél alá gyermekként beszúrjuk az új csúcsot.

Adott n számú pont esetén megmutatható, hogy a keresés és a beszúrás átlagos költsége $O(\log_2 n)$ lesz (*Jason Scott Bentley* /1975/). A fa felépítésére fordított költség megegyezik a fa teljes úthosszával (*TPL = Total Path Length*, összes út hosszának összege), azaz az összes pont lekérdezésére fordított költséggel. n számú, véletlenszerűen elhelyezkedő pontból épített kd-fa teljes úthossza $O(n * \log_2 n)$. Ebből következik, hogy a csúcsbeszúrás és keresés átlagos költsége $O(\log_2 n)$. Ennél persze sokkal rosszabb esetek is előfordulhatnak, mert a fa szerkezete erősen függ a beszúrt pontok sorrendjétől, alakja nagyon megnyúlhat, a fa kiegyensúlyozatlanná válhat. Ennek következménye, hogy a teljes úthossz megnőhet. A tartománylekérdezések költsége a legrosszabb esetben $O(k * n^{1-1/k})$.

Egy pont törlése nehezebb eset, mert nem biztos, hogy egy kd-fa részfája is kd-fa lesz, így a rekurzív törlő algoritmus végrehajtása után a fa újraszervezésére is szükség lehet, ami további plusz költséget jelent. A törlés azért nem egyértelmű, mert minden szinten más koordináta kerül összehasonlításra, mint az előző szinten, így minimum kereséskor nem feltétlen mindig a baloldali részfákban kell keresni, ahogy azt a bináris keresőfáknál egyébként megszoktuk. Levelek törlésekor az átlagos költség felülről becsülhető $O(\log_2 n)$ -nel, azonban egy gyökércsúcs törlése lényegesen nagyobb műveletigénnyel jár. Egy részfa gyökerének törlésekor felmerülő költség felülről korlátos a részfa csúcsainak számával. Ha a csúcsok száma n és figyelembe vesszük, hogy csúcstörléskor szükséges egy D-minimum elem keresése, amelynek

műveletigénye $O(n^{1-1/k})$, akkor ez a költség fog dominálni egy csúcs törlésekor is. Ez annak a következménye, hogy minden *k*-adik szinten a két részfa közül csak az egyikben szükséges tovább keresni.

A kiegyensúlyozatlanság mellett a fa másik hátránya, hogy az egyes hípersíkok az általuk metszett pontokkal kerülnek azonosításra, de előfordulhat, hogy a fa optimálisabb alakjához és a pontok egyenletesebb eloszlásához más azonosító pontra lenne szükség.

4.3 Adaptív kd-fa

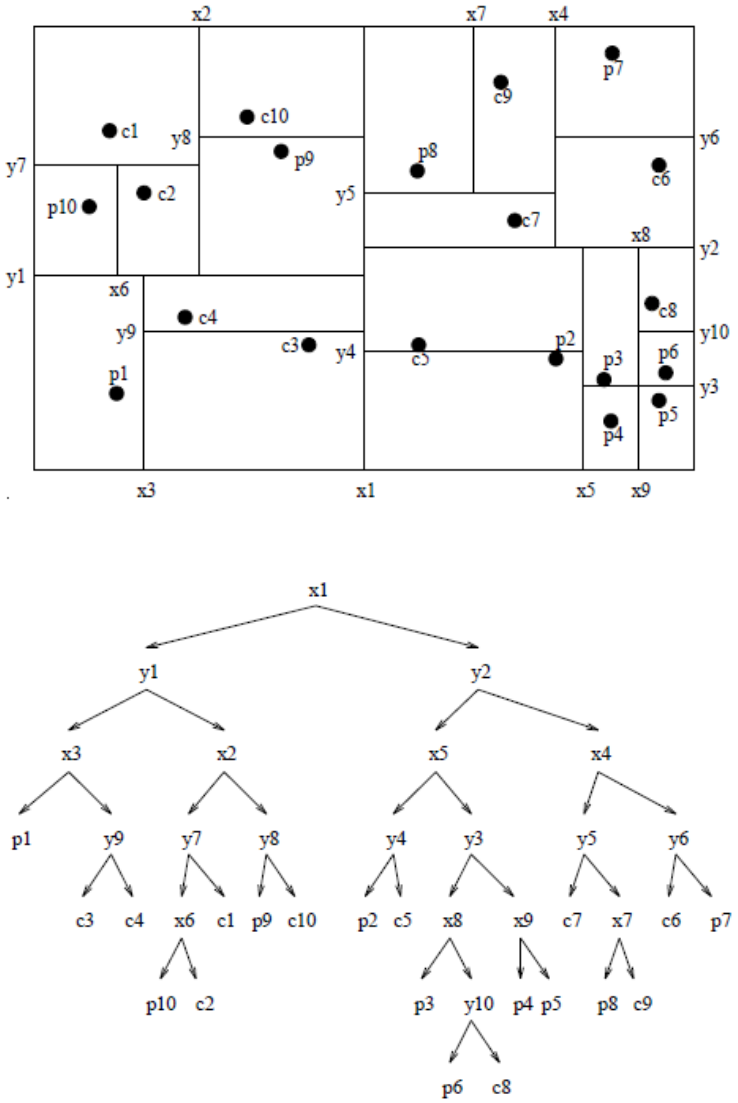
A kd-fák legnagyobb hátrányát az okozza, hogy a felépített fa az objektumok elhelyezkedésétől függően nagyon kiegyensúlyozatlanná is válhat. Alakjuk erősen függ a beszúrt pontok sorrendjétől, így az összes út hosszától. Tárhely igényük a megnyúlt méretük miatt megnőhet illetve a bennük való keresések időben elhúzódhatnak, növelve ezzel a költségeket. A *TPL* csökkentésére egy statikus és egy dinamikus módszer is létezik.

A dinamikus módszer lényege, hogy a fa építése az általános esetben leírtaknak megfelelő módon történik, de ha az építés során egy előre definiált egyensúlyi határt átlép, akkor a fát részben átrendezzük, hogy az egyensúlya megmaradjon.

A statikus módszer lényege, hogy a térben elhelyezkedő pontok beszúrásának sorrendjét előre ismerjük. Erre nyújt megoldást az adaptív kd-fa.

Adaptív esetben a tér felosztásakor nem feltétel az, hogy a hipersík és valamely input pont metszésben legyenek. Helyette olyan hipersíkot választunk a vágásra, amely a teret úgy osztja két részre, hogy a keletkező két térrészben a pontok eloszlása nagyjából egyenlő legyen [6].

A hípersíkok továbbra is az alterekkel párhuzamosak, de itt már nem elvárás, hogy valamely pontot tartalmazzák. Ennek eredményeképpen, az input pontokat nem a belső csúcsokban tároljuk, hanem csak a levelekben fognak megjelenni. A belső csúcsokban a hipersíkok azonosítója szerepel. Ez az azonosító a következő két elemből áll: egy dimenzió vagy tengely elnevezés (pl. *x* vagy *y*), amellyel párhuzamosan történt a vágás és egy koordináta, amely általában a keletkezett két térrészben elhelyezkedő objektumok megfelelő koordinátájának valamilyen statisztikai közepe (pl. mediánja). A rekurzív darabolást addig végezzük, amíg a keletkezett részekben adott számú objektum nem marad (4.10 ábra).



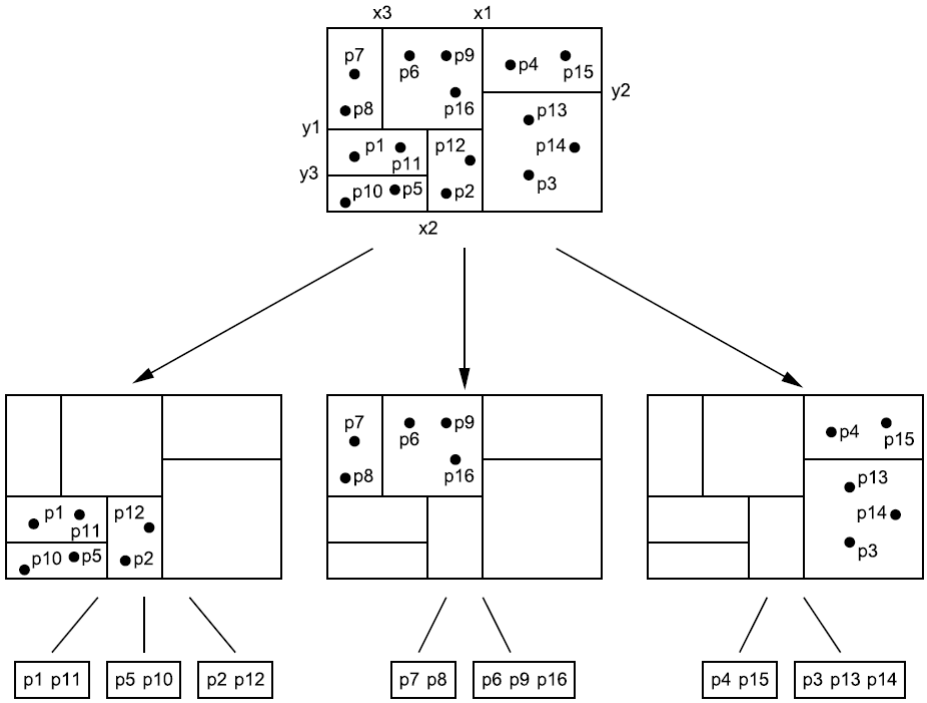
4.10 ábra: adaptív kd-fa

n számú csúcsból felépített adaptív kd-fa építési költsége a teljes úthosszból adódóan $O(n * \log_2 n)$. Az adaptív kd-fa egy statikus szerkezet olyan értelemben, hogy felépítéséhez a térbeli objektumok számát és a térben való elhelyezkedésüket előre ismernünk kell. Alakja továbbra is függ a beszúrt pontok sorrendjétől. Nem feltétlen lesz a fa kiegyensúlyozott, de az alakzatok elhelyezkedését ismerve nagyobb a valószínűsége, mint a kd-fák esetén. Előnye, hogy a keresések felgyorsulhatnak. Hátránya, hogy kiegyensúlyozottság esetén e tulajdonság fenntartása beszúráskor és törléskor nehézséget jelent és külön költséggel jár. Leghatékonyabban akkor alkalmazható, ha nem sűrűn hajtunk végre adatmódosító műveleteket az adatbázison, azaz a feltételes és tartományi lekérdezésekre használható jól. Az utóbbiak műveletigénye 2D esetén $O(\sqrt{n} + t)$, ahol *t* a megtalált pontok számát jelöli.

4.4 kd-B-fa

A kd-B-fa (*k-dimensional B-tree*) az adaptív kd-fa és a B-fa tulajdonságait egyesíti. A tér felosztásához továbbra is az adaptív esetnél látott hipersíkokat alkalmazza, de a reprezentációhoz az egydimenziós indexeléseknél bemutatott B-fát használja.

Ennek a struktúrának az előnye, hogy egy csúcsába egyszerre több hipersík azonosítója is bekerülhet, felépítése tömörebb, kevesebb a helyigénye és a B-fák teljesen kiegyensúlyozott tulajdonságát örökli. A pontok eloszlása egyenletessé válik. A fa minden egyes belső csúcsa a háttértároló egy blokkjának felel meg. Az egy szinten lévő csúcsok egymáshoz viszonyítva diszjunktak és uniójuk kiadja az egész teret. Az input pontok ismét csak a levelekben fognak megjelenni. Egy levélben adott számú pont fér el. Ha a pontok száma meghaladja az adott korlátot, akkor további térfelosztásra van szükség [6] (4.11 ábra).



4.11 ábra: kd-B-fa

Egy térbeli objektum keresésekor a fa gyökerétől kell eljutni a megfelelő levélig oly módon, hogy minden egyes csúcsban megvizsgáljuk, hogy a keresett objektum melyik térrészbe esik bele és ennek megfelelően lépünk tovább a fa következő szintjére. A kereséshez felhasznált I/O műveletek száma megegyezik a fa szintjeinek számával, hiszen minden szinten a csúcsok egy blokknak felelnek meg és kereséskor a

szomszédos csúcsok diszjunkt tulajdonsága miatt, minden szintről kizárólag egy csúcsban elhelyezkedő térrész kerül beolvasva.

A B-fákhoz hasonló módon a lekérdezések felgyorsulnak, de beszúráskor és törléskor a beolvasásokon kívül felmerülnek adatmanipulációs költségek is (csúcsvágás és csúcsösszevonás). A csúcsvágáshoz hozzátartozik annak vizsgálata, hogy a csúcs telített-e már és valóban szükséges-e a vágás. Ezt követően ki kell választani az optimális vágási és pont elosztási módszert. Csúcsok összevonása előtt szükséges ellenőrizni, hogy marad-e a csúcsban elegendő mennyiségű pont, vagy összevonást kell alkalmazni. A fa kiegyensúlyozottsága miatt mindkét adatkezelő művelet maga után vonhatja az egész fa újraszervezését, ami újabb költségekkel jár, csökkentve ezzel a hatékonyságot.

4.5 BSP-fa

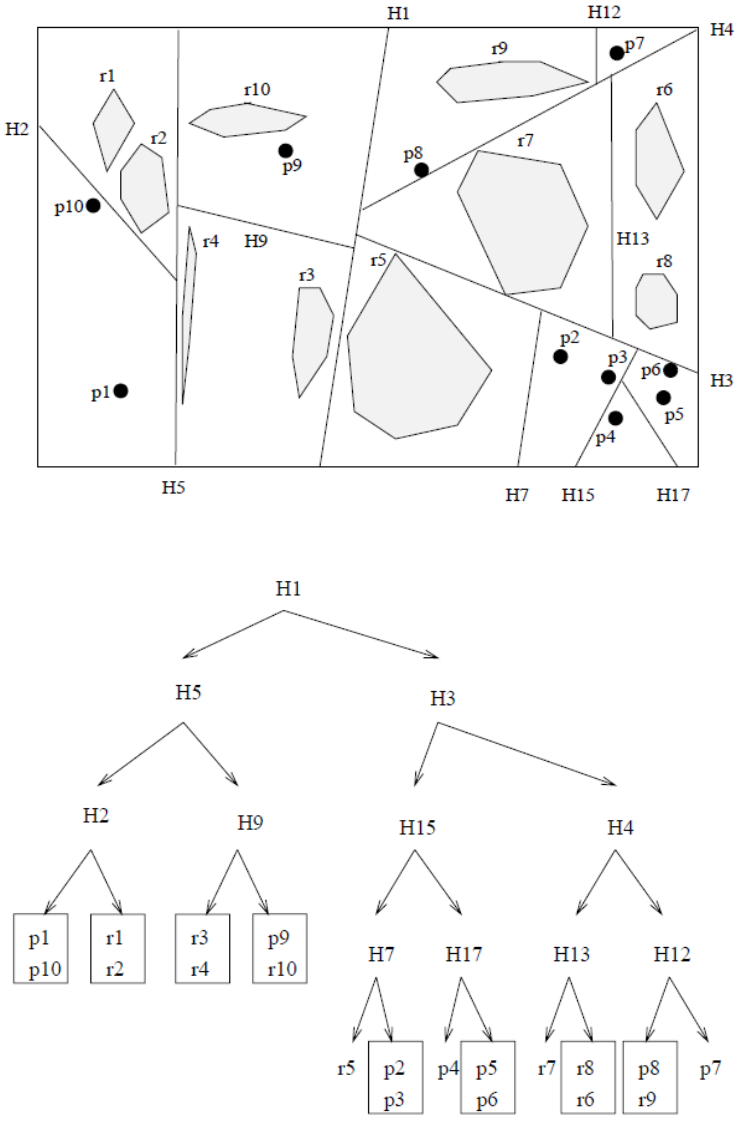
Az előző alfejezetekben bemutatott kd-fák hátránya, hogy bizonyos elhelyezkedésű ponthalmazok esetén nem található optimálisan vágó hipersík a térben. Egy sokkal rugalmasabb partícionálási sémát bevezetve a BSP-fa (*Binary Space Partitioning tree*) hatékony megoldást nyújt erre a problémára. A rugalmasságot az adja, hogy nincs megkötve a vágósíkok valamely tengellyel való párhuzamossága, így a térfelosztás elvégezhető úgy, hogy a pontok eloszlása a lehető legegyenletesebb legyen. Ennek eredményeképpen a térfelosztások nem függenek a korábban végrehajtott vágásoktól, csak az adott térrészben elhelyezkedő objektumok pozíciójától.

A BSP-fa olyan bináris keresőfa, amely a d dimenziós tér, d - 1 dimenziós hipersíkok segítségével, konvex alterekre való rekurzív felosztását reprezentálja. A térfelosztást addig végezzük, amíg a keletkezett részekben egy megengedett számú objektum nem marad. Ez a megengedett szám a háttértároló blokkméretétől függ. A tér ilyen módon történő partícionálásának eredménye egy BSP-fával a következő módon ábrázolható: a fa belső csúcsaiban a vágásokra vonatkozó információk és a szeparáló síkok találhatók, a levelekben az input objektumok szerepelnek (4.12 ábra).

Pont keresése a gyökérből kiindulva – hasonlóan a kd-fáknál bemutatott algoritmussal – történik. Minden csúcsban a megfelelő koordináták összehasonlításával arról döntünk, hogy a csúcsban szereplő hipersík melyik oldalán található a keresett pont és ennek megfelelően lépünk tovább a fa következő szintjére. Bármely levél elérése után, megvizsgáljuk, hogy az abban hivatkozott objektumok közül melyik

tartalmazza a pontot vagy melyik maga a pont. A keresés műveletigénye $O(n)$ ahol n a fa elemeinek száma. A területlekérdezések a pontkeresés módszerének általánosításai.

A BSP-fa az egyik leghatékonyabb eszköz térbeli objektumok indexelésére. Jól alkalmazható a tér több különböző módon való felosztására. Bár az algoritmus helyes működése nem függ attól, hogy hogyan választjuk meg a szeparáló síkjainkat, a síkok által végrehajtandó vágások száma azonban jelentősen befolyásolja az algoritmus időigényét. Hátrányuk, hogy nem kiegyensúlyozottak, továbbá nem optimális hipersík mentén való térfelosztáskor lehetnek benne elég hosszúra nyúlt részfák is. Tárolásuk több helyet igényel, mint a korábban bemutatott fa struktúrák bármelyike, mert korábban a belső csúcsokban csak egy hipersík azonosító került tárolásra, de BSP-fák esetén a vágósíkot leíró összefüggések is szerepelnek.



4.12 ábra: BSP-fa

4.6 Térkitöltő vonalak

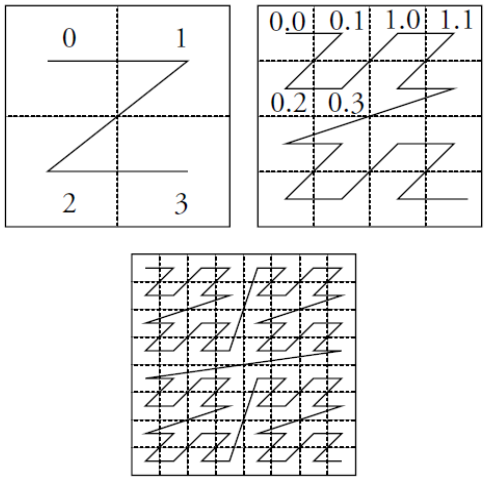
Ebben az alfejezetben a térkitöltő vonalakat, tulajdonságaikat és felhasználásaikat mutatom be, mert a következő indexelési módszerek alkalmazásában alapvető szerepet játszanak.

Korábban említettem, hogy nehéz megfeleltetést találni az egydimenziós és a többdimenziós adatok között, mert nem létezik a többdimenziós térnek távolságtartó leképezése egydimenziósra. A cél mégis olyan leképezést adni, amely legalább bizonyos mértékben megőrzi a térbeli objektumok egymáshoz való közelségét. Ha ilyen módon sikerül egydimenziósra leképezni a teret, akkor alkalmazhatók lesznek a második fejezetben bemutatott egyszerű indexelési eljárások. Erre nyújtanak megoldást a térkitöltő vonalak.

Ha a többdimenziós teret cellákra osztjuk fel, az egyik dimenzió mentén, majd vesszük valamely térkitöltő vonalat, akkor ezek a vonalak a 2D rácsoknak egy egyértelmű sorrendjét adják meg. Ezzel a többdimenziós teret leképezzük az egydimenziós térre úgy, hogy az eredeti térben egymás közelében lévő objektumok a rendezést követően is viszonylag közel lesznek egymáshoz. A két leggyakrabban használt térkitöltő vonal a Z-vonal és a Hilbert-görbe.

4.6.1 Z-vonal

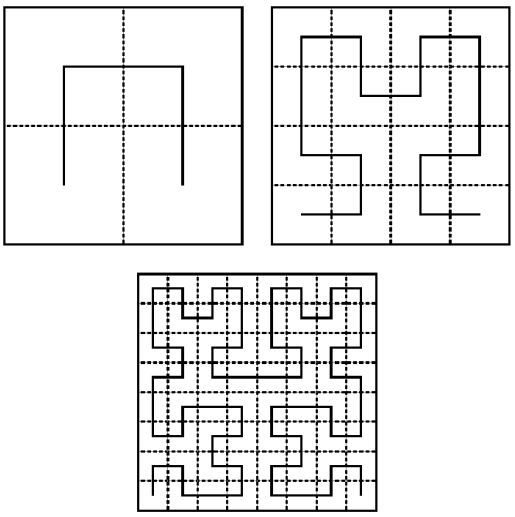
A leképezés végrehajtásához létrehozott cellákat számokból álló címkékkel látjuk el a következő módon. A teljes teret négy egyenlő méretű négyzetrácsra osztjuk fel, és mindegyik cella kap egy $c \in [\,0..3\,]$ címkét. Az így kapott cellákat, ha szükséges, további négy rácsra osztjuk fel és az új cellákat is címkékkel látjuk el. Ezek a címkék a *c*.0, *c*.1, *c*.2 és *c*.3 lesznek, ahol a pont a konkatenációt jelenti. Ezt a rekurzív negyedelő eljárást addig folytatjuk, amíg a cellák valamelyike telített állapotban van (bővebben lásd: negyedelő fáknál), azaz a megengedettnél több objektumot tartalmaz. A keletkezett cellák a címkéjük szerint lexikografikusan rendezhetők. A rendezett cellák középpontját NW (észak-nyugat), NE (észak-kelet), SW (dél-nyugat), SE (dél-kelet) irányban összekötve kapjuk a Z-vonalat. Minden negyedelés után egy $N \times N$ -es méretű mátrix jön létre, ahol $N = 2^d$. A d hatvány a negyedelés mélységét jelenti, azaz a cellákhoz rendelt címkék d hosszúságúak lesznek. Hátránya, hogy ha a nem szomszédos cellák kerülnek összekötésre, akkor ennek eredményeképpen nagy ugrások adódnak. A 4.13 ábrán látható a Z-vonal $d = 1$, $d = 2$ és a $d = 3$ esete.



4.13 ábra: Z-vonalak (Morton kód [1])

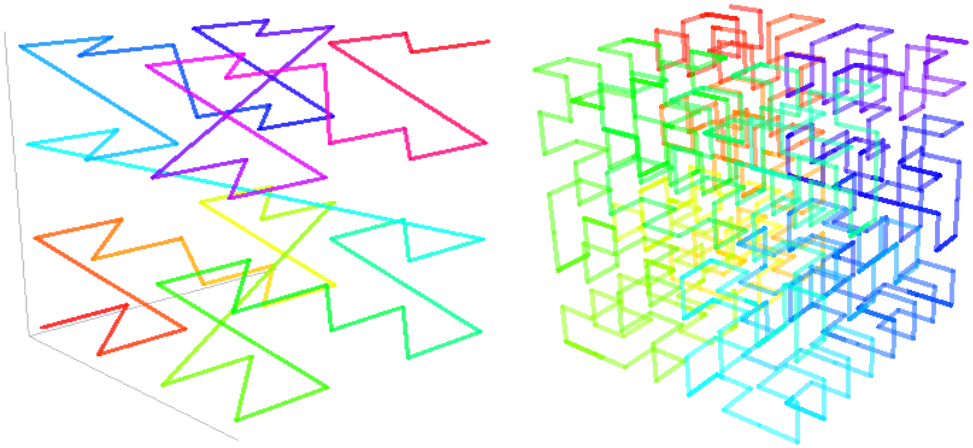
4.6.2 Hilbert-görbe

A Hilbert-görbe a Z-vonalhoz teljesen hasonló módon hozható létre. A különbség a cellák összekötési sorrendjében van. Kizárólag szomszédos rácsok kerülnek összekötésre, ezzel kiküszöbölhető a Z-vonalaknál látott nem szomszédos cellák közötti előnytelen nagy ugrás. A térkitöltő vonalak alakja nem *Z* lesz, hanem *Π*. A 4.13 ábrán a Hilbert-görbe $d = 1$, $d = 2$ és $d = 3$ esete látható.



4.14 ábra: Hilbert-görbék

Mindkét görbénél észrevehető, hogy adódhatnak olyan esetek, amikor két objektum a 2D térben közel van egymáshoz, de a térkitöltő vonal mentén haladva távol kerülnek.



4.15 ábra: Z-vonalak és Hilbert-görbék 3D esetben [13] [14]

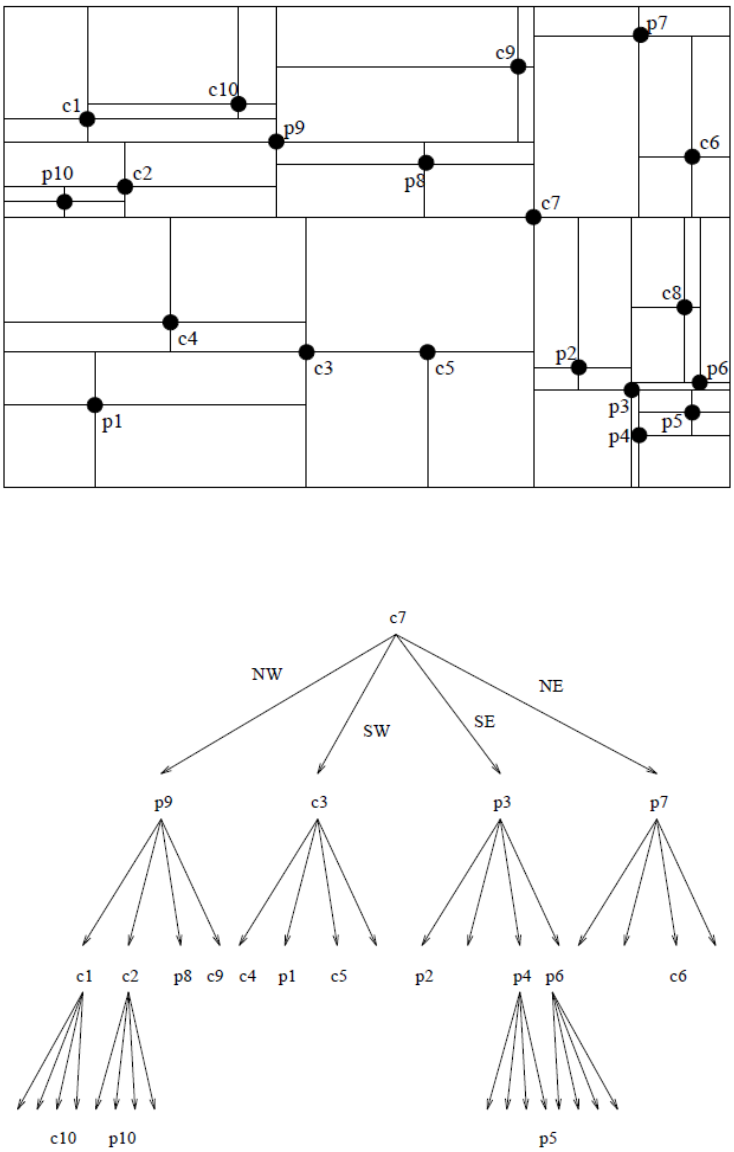
4.7 Negyedelő-fa

A negyedelő-fát (*quadtree*), más elnevezéssel négyfát, használják a leggyakrabban a térben elhelyezkedő objektumok indexelésére. Ennek oka az egyszerűségében rejlik. Speciálisan a 2D térben nevezik negyedelő-fának, de több dimenzió esetén is alkalmazható. Ez az elnevezés egy általános megnevezése azoknak a fáknak, amelyek úgy épülnek fel, hogy a síkot mindig négy részre, speciális esetben negyedekre bontjuk fel. Ha a dimenzió d és $d > 2$, akkor a térfelosztáskor 2^d tartomány jön létre. Általánosan mondható, hogy egy d dimenziós térben használva a negyedelő-fát, minden csúcsának, amely nem levél, 2^d számú leszármazottja lesz. A negyedelő-fa háromdimenziós kiterjesztését nyolcfának (*octtree*) nevezzük. A kd-fákhoz hasonlóan, a térfelosztás egy rekurzív algoritmussal történik. Többféle változatuk létezik, ezek közül a legnépszerűbbek a következők: pont négyfa (*Point quadtree*), a ponttartomány négyfa (*Point Region (PR)-quadtree*) és a tartomány négyfa (*Region quadtree*).

4.7.1 Pont négyfa

A pont négyfa elnevezés arra utal, hogy a csúcsokban pontokat tárolunk, azaz pontszerű objektumok indexelésére alkalmazható. Összetett objektumok indexelésekor ismét a *centroid*-jukat vesszük alapul. A négyfák pont alapú típusa a bináris keresőfának olyan többdimenziós általánosítása, amelyet ugyanúgy építünk fel, mint az általános kd-fákat, azzal a különbséggel, hogy nem két, hanem mindig négy részre osztjuk a teret. Az adatpontokat metsző, a koordináta tengelyekkel párhuzamos, egymásra merőleges egyeneseket véve, kialakul a négy térrész és felépül a fa. A térfelosztást rekurzív módon addig ismételjük, amíg van szabad input pont a térben, amelyet még egyetlen hipersík

sem metsz el. Minden csúcsban az input pontokat tároljuk és mindegyiknek négy gyermeke (mutatók) van, NW, NE, SW és SE címkékkel ellátva. A címkék azt jelzik, hogy a csúcs gyermeke a szülőben tárolt ponthoz képest melyik térnegyedben helyezkedik el [6] (4.16 ábra).



4.16 ábra: pont negyedelő-fa

Pont keresése és beszúrása a kd-fáknál bemutatott algoritmussal történik. A különbség az, hogy az algoritmus minden szinten mindkét koordinátát összehasonlítja és ennek megfelelően lép tovább NW, NE, SW vagy SE címkéjű irányba. Ez annak a következménye, hogy térfelosztásra nem csak az egyik irányú hipersíkot vesszük, hanem negyedelést végzünk. A *TPL*-től függően és figyelembe véve, hogy minden

csúcsnak négy gyermeke van, a műveletekre fordított költségek is hasonlóan alakulnak, mint a kd-fák esetében. Ha a fát *n* számú pontból építjük fel, akkor a keresés és beszúrás átlagos költsége $O(\log_4 n)$. A törlés művelete a térrészek megnövekedett száma miatt bonyolulttá válik.

Szerkezete erősen függ a beszúrt pontok sorrendjétől, így kedvezőtlen esetben nagyon megnyúlhat. Ha az input pontok eloszlása nem túl egyenletes, akkor mérete a sok felosztás miatt megnő. Ahogy a kd-fáknál, itt is adott egy statikus és egy dinamikus módszer az optimalizálásra. Statikus esetben a pontok sorrendjét előre ismerjük (lásd Adaptív kd-fa módszere). Dinamikus esetben a fa építésekor kiegyensúlyozottságot segítő átrendezéseket végzünk.

Tipikusan az objektumok közelségét vizsgáló tartományi lekérdezésekre használják ezt a szerkezetet. Pl.: Azoknak a városoknak az összességére vagyunk kíváncsiak, amelyek Budapest 50 km-es környezetébe esnek. A tartománylekérdezések költsége *k* dimenzió esetén, a legrosszabb esetben $O(k * n^{1-1/k})$.

4.7.2 Ponttartomány négyfa

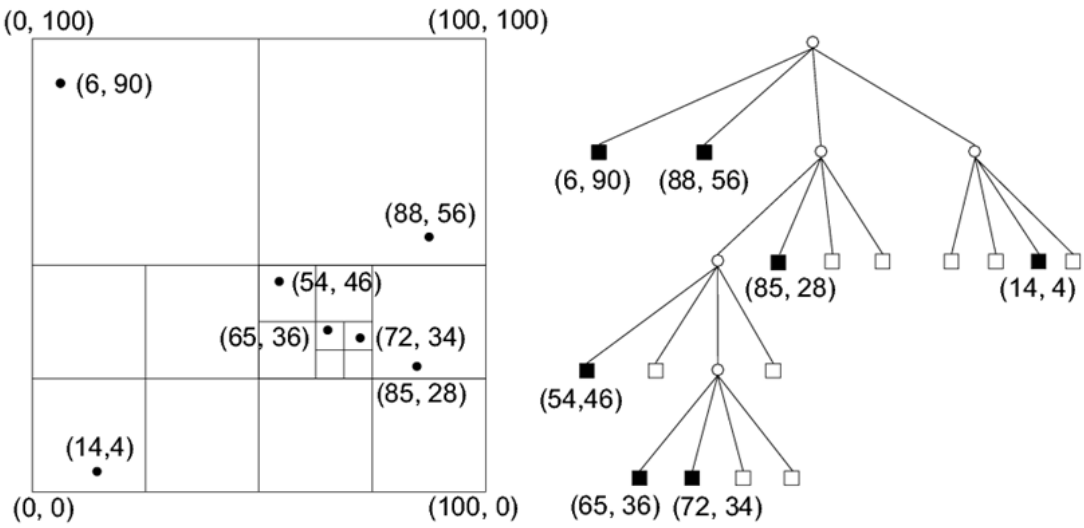
Ez a negyedelő-fák leggyakrabban használt típusa. Jól alkalmazható összetett objektumok indexelésére is, mert az objektumokat a *centroid*-juk mellett az *mbb*-jükkel is azonosíthatjuk. Az utóbbi esetben a téglalapokra vonatkozó rész kiválóan megfelel ezen összetett alakzatok tárolására. Ponttartomány négyfák esetén a teret mindig négy egyenlő részre osztjuk fel, és így építjük fel a fát.

Pontok indexelése [1]: A térrészek egyenlő felosztásából adódóan nem elvárás, hogy a felosztáshoz használt hipersík metszésben legyen valamely input ponttal. A rekurzív negyedelő eljárást addig folytatjuk, amíg el nem érjük, hogy minden létrejött tartomány legfeljebb egy input objektumot tartalmaz.

A fa belső csúcsai a negyedeléssel keletkező tartományokat reprezentálják, leveleiben az input pontok találhatók. A fa gyökere megfelel a kiindulási térnek, amelyben az objektumok elhelyezkednek. A belső csúcsokban az egyes térrészek reprezentációs módja a következőképpen alakul: a csúcs tartalmazza a hozzá tartozó térrész felosztásának középpontját azonosító koordinátákat; illetve négy mutatót, amelyek a felosztás után keletkező négy új tartományra mutatnak. Ezeknek a jelölésére a következőket használjuk: NW, NE, SW és SE. A fa levelei lehetnek üresek (fehér

színűek) vagy egy input pontot tartalmazhatnak (fekete színűek). Minden levél a háttértároló egy blokkjának felel meg [5] (4.17 ábra).

A fa felépítése az egyes pontok egymás utáni beszúrását jelenti. A gyökértől egy kereső eljárással addig haladunk lefelé, az egyes szinteken a koordinátákat összehasonlítva, amíg egy levélhez nem érünk. Ha a levél üres, akkor a pontot eltároljuk benne. Ha a levél nem üres, akkor csúcsvágásra van szükség, azaz az aktuális tartományt újabb és újabb négy részre osztjuk. A negyedelés addig folytatódik, amíg a régi és az új csúcs külön tartományba nem esik. A beszúrás után a két pont egymás testvére lesz. Ha a pontok túl közel vannak egymáshoz, akkor a negyedelés egymás után többször ismétlődik, ezáltal a fa alakja megnyúlik.



4.17 ábra: ponttartomány negyedelő-fa: pontok indexelése

A szabályos és fix negyedelések miatt, a struktúra szerkezete nem függ a beszúrt pontok sorrendjétől, azonban az eloszlásuk meghatározó szerepű a fa alakjában és méretében. Sűrűn elhelyezkedő ponthalmazok esetén a sok negyedelés miatt, a fa alakja megnyúlik. Minden negyedelés alkalmával ugyanis egy meglévő levélnek újabb négy gyerekcsúcsa jön létre. A keresések, a fa magassága miatt, időben elhúzódnak, az adatmanipulációs műveletek költségei megnőnek.

Pont keresésekor a fa gyökerétől kell eljutni a megfelelő levélig oly módon, hogy minden egyes csúcsban megvizsgáljuk, hogy a keresett objektum melyik térnegyedbe esik és ennek megfelelően lépünk tovább a következő szintre. Ahogy a fa felépítésénél láttuk, beszúráskor csúcsvágásra lehet szükség, attól függően, hogy a pontok milyen

sűrűn helyezkednek el a térben. Hasonlóan, ha egy pont törlése után egyetlen levele marad a csúcsnak, akkor a résztartományokat érdemes összevonni egybe.

Látható, hogy a műveletek futásideje, ahogy a korábbi esetekben is, a fa maximális mélységétől függenek. *n* csúcspont esetén a minimális mélység $\lceil \log_4(n-1) \rceil$. Ekkor a fa kiegyensúlyozott. Ez persze az optimális eset, általánosságban nem mondható igaznak. Felső korlát azonban nem adható ilyen könnyen, kivéve, ha a következő optimalizációt elvégezzük: a fa alakja tömörebbé tehető, ha olyan változatát használjuk, ahol a tartományok felosztásának feltételét a blokk méretéhez igazítjuk, azaz csak abban az esetben osztunk tovább egy térrészt, amikor egy adott csúcshoz tartozó lap a megnövekedett számú pontok miatt túlcsordul. Ilyenkor bármely levelet elérve további keresésre van szükség az adott blokkban a pontos eredmény meghatározásához. Mivel a mélység a pontok eloszlásától függ, ezért egy másik korlátozó lehetőség a megnyúlásra, ha feltesszük, hogy a vizsgált térrész kiterjedése *t* és a benne elhelyezkedő objektumok közül bármely kettőnek az Euklideszi távolsága legalább *d*. Legrosszabb esetben egy *d* hosszú szakasz, valamely koordináta tengelyre vett vetülete $d/\sqrt{2}$ hosszú. Ekkor a *t* kiterjedésű tartományba maximum $t / \left(d / \sqrt{2} \right)$ ilyen szakasz fér el. Így a mélységre a következő felső korlát adható: $\lceil \log_2\left(\left(t / d \right) * \sqrt{2} \right) \rceil$

A tárolás költségéről a következő megállapítást tehetjük. Ha feltételezzük, hogy az input objektumok elhelyezkedése a térben lefedhető egy $2^N \times 2^N$ méretű négyzetráccsal, akkor *n* számú input objektumból épített fa tárolási igénye legfeljebb $O(n * N)$ lesz.

A négyfák is támogatják azokat a tartományi lekérdezéseket, amelyek téglalapba eső pontokat keresnek. A téglalap oldalai a negyedelő síkokkal párhuzamosak. Az ilyen típusú keresések műveletigénye legrosszabb esetben $O(t * 2^N)$, ahol *t* a megtalált pontok száma és *N* a fa maximális mélysége.

Téglalapok indexelése [1]: Ebben a részben megvizsgálom, hogyan lehet a ponttartomány négyfával téglalapokat indexelni. Ha a komplex alakzatainkat a legkisebb befoglaló téglalapjukkal azonosítjuk, akkor ennek a módszernek a segítségével indexelésük visszavezethető erre az esetre.

A fa felépítése ugyanúgy történik, mint a pontok indexelésekor. Alakzat beszúrásakor a negyedelést addig végezzük, amíg minden tartományban legfeljebb egy adott számú objektum nem marad. Ez az adott szám a blokk méretének felel meg.

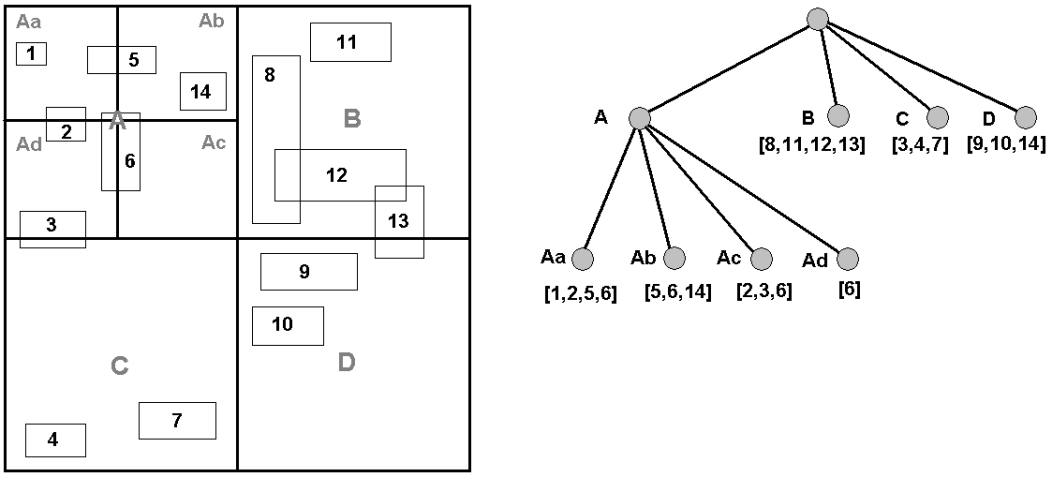
Azt mondjuk, hogy egy *t* tartomány fedésben van egy *mbb*-vel, ha az alábbi három eset közül az egyik teljesül:

(1) a téglalap tartalmazza a tartományt

(2) a tartomány tartalmazza a téglalapot

(3) a tartomány és a téglalap metszik egymást

Ekkor az *mbb* minden, vele fedésben lévő tartományt reprezentáló levélben meg fog jelenni az azonosítójával (4.18 ábra). Új téglalap beszúrása és törlése ugyanúgy történik, mint a pontok esetén, azzal a kivétellel, hogy (1) és (2) esetben lehetséges, hogy a végrehajtott művelet több levelet is érinteni fog, ezzel a költségeket növelve. Akár pont, akár téglalap beszúrása után, ha valamelyik blokk túlcsordul, akkor további negyedelések (csúcsvágások), törléskor pedig tartomány összevonások fordulhatnak elő.



4.18 ábra: ponttartomány negyedelő-fa: téglalapok indexelése

Pontszerű objektumok keresésének, beszúrásának és törlésének algoritmusa teljesen megegyezik a pont négyfánál alkalmazott módszerekkel.

Téglalapok keresésekor a gyökérből kiindulva, minden olyan levélig el kell jutni, amely által reprezentált tartomány a keresett téglalappal fedésben van. Ennek eredményeképpen lehet, hogy a fában több utat is be kell járni az eredmény meghatározásához.

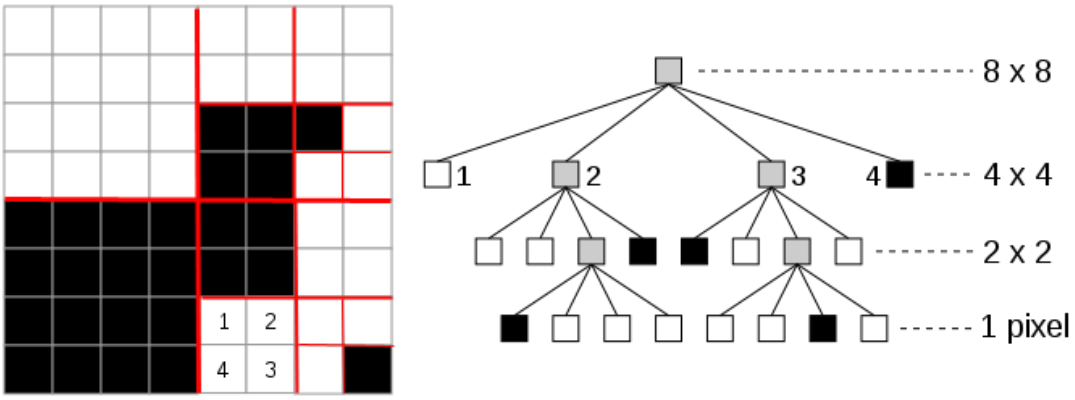
Sűrűn elhelyezkedő téglalapok indexelésekor a túlcsordulás valószínűsége nagyobb, további darabolásokra van szükség, ez az index méretének növekedésével és a hatékonyság csökkenésével jár. A téglalapok mérete is befolyásoló tényező, hiszen ettől függ, hogy hány levélben szükséges a téglalapot eltárolni, ami megmutatja, hogy kereséskor hány utat kell bejárni a fában. Minél nagyobb a téglalap, annál nagyobb az

esélye, hogy több tartományt is le fog fedni. Ezáltal, a grid indexelési módszernél is látott, duplikációk lépnek fel. A duplikátumok száma a negyedelések többszöri alkalmazásával exponenciálisan nő, ezáltal a keresések és a törléseket lelassulnak, a tárigény megnövekszik, és az index elveszti hatékonyságát.

4.7.3 Tartomány négyfa

A negyedelő-fák tartományokra vonatkozó típusát nem részletezem teljes mélységében, mert általában nem a vektoros modellre – amely a dolgozatom középpontjában áll – hanem a raszteres képek indexelésére alkalmazzák. Érdekességképpen említem csak meg ennek folyamatát és az eredményt egy ábrán is szemléltetve.

A fa felépítéséhez a megszokott negyedeléseket végezzük el. A negyedelések számára tehetők különböző megkötések, hogy ne pixelnyi méretekig kelljen elmenni. Pl.: Egy fekete-fehér képet véve alapul, térfelosztásokat addig folytatjuk, amíg az egyes tartományi negyedekben csak azonos színű pixelek maradnak (4.19 ábra). Ugyanezt szolgálja, ha feketével jelöljük a térképi alakzatunk belső pixeleit és fehérrel azokat, amelyek nem tartoznak hozzá az alakzathoz. A módszer különböző felbontású képekre is alkalmazható.

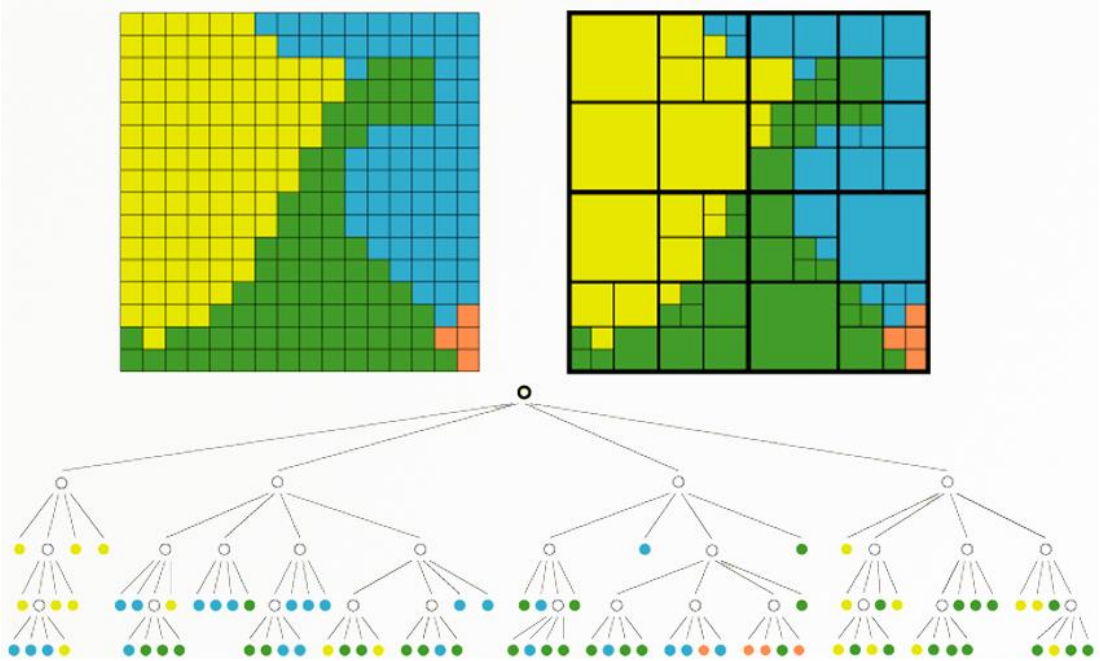


4.19 ábra: tartomány negyedelő-fa: fekete-fehér raszteres képek indexelése

Az ábrán látható fában a keresések, a beszúrások és a törlések algoritmusa a korábbiaknak megfelelően működik. Objektum keresésekor az azonos színű pixelek helyét kell megtalálnunk a fában. Ez a fenti példában azokat a leveleket fogja érinteni, amelyekben a fekete színű pixelek kerültek tárolásra.

Az eljárás kiterjeszthető több színű képekre is. Színes képekre két általános alkalmazást adok meg. A két megoldás a negyedelés feltételében különbözik egymástól:

- az egyes tartományok méretét nem egy konkrét pixel színe határozza meg, hanem több, egymástól csak árnyalatukban eltérő pixelek összessége. Ez azt jelenti, hogy ha a közel azonos intenzitásértékű pixelek egy tartományba esnek, nem végzünk további negyedeléseket. A felosztást csak akkor folytatjuk, ha az értékek között egy adott korlátnál nagyobb az eltérés. Pl. erre lehet egy tól-ig intervallumot megadni.
- jóval bonyolultabb eljárások (pl. klaszterezés, tematikus térképek készítése) után, a kialakult tematikus tartományokba tartozó képpontok foglalnak helyet a fa levelein (4.20 ábra).



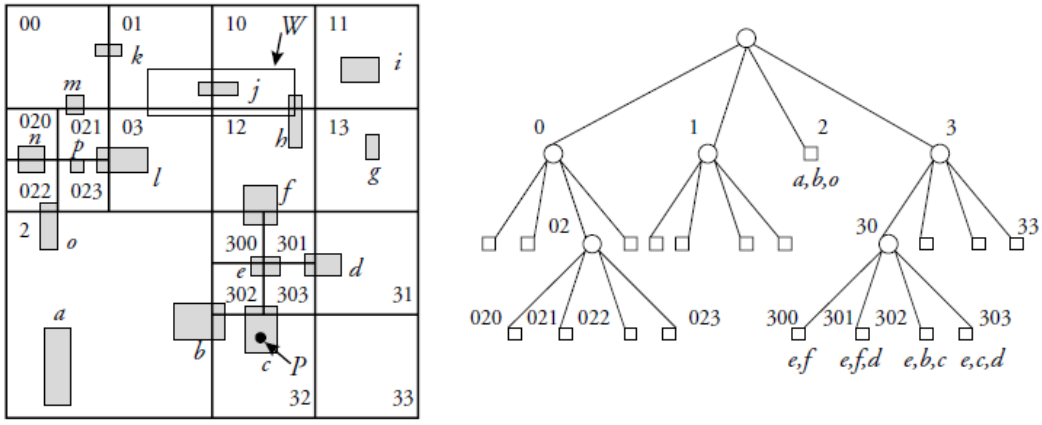
4.20 ábra: tartomány negyedelő-fa: színes raszteres képek indexelése

A negyedelő-fák eddigi verzióiból jól látható, hogy tárolásuk nem olyan hatékony, mint a B-fáké. Ennek oka, hogy a B-fákkal ellentétben, minden csúcsnak csak négy gyereke lehet, amely a blokkok kihasználtságának szempontjából nem túl előnyös, mert azok többsége csak részben lesz telített vagy teljesen üresen marad. Nehéz leképezést találni a négyfa és a lemez blokkjai között. Sokkal hatékonyabbak ilyen szempontból a B-fák és a következő fejezetben szereplő R-fa és egyes típusai. Ha a legrosszabb esetet vizsgáljuk, akkor minden belső csúcs másik lapra kerül. Kereséskor a fa minden szintjén más blokkot kell beolvasnunk, így az I/O műveletek száma meg fog egyezni a fa mélységével. Láttuk, hogy a fa magassága bizonyos esetekben nagyon megnyúlhat.

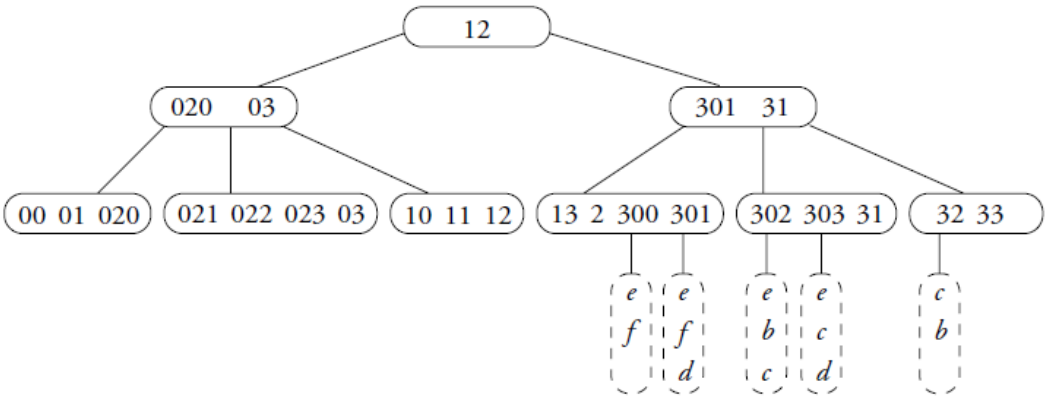
4.8 Lineáris negyedelő-fa

Felhasználva a térkitöltő vonalakat bevezethetünk egy, a blokkokat optimálisabban kihasználó struktúrát, amelyet lineáris negyedelő-fának nevezünk.

Az alapötlet az, hogy miután a négyfa felépült, a térkitöltő vonalaknál látott eljárást alkalmazva, minden levélhez hozzárendelhetünk egy őt azonosító címkét, amelynek hossza megegyezik a gyökértől az adott levélig vezető út hosszával. Ha a Z-vonalnál látott (NW, NE, SW, SE) sorrendnek megfelelően címkézzük fel a leveleket, akkor balról jobbra végig haladva rajtuk, a címkéknek lexikografikusan növekvő sorozatát kapjuk (4.21 ábra). Ebben a sorozatban nem minden levél címkéje lesz azonos hosszú. Mindezek után, a levélcímkék rendezettségét kihasználva, egy B^+-fa indexet építhetünk fel a levelekből kiindulva (4.22 ábra). Ezzel a módszerrel több levél elhelyezhető egy blokkban, így a helykihasználás optimálisabbá tehető.



4.21 ábra: negyedelő-fa leveleinek címkézése Z-rendezéssel



4.22 ábra: B^+-fa a negyedelő-fa leveleinek címkéjéből

Lineáris negyedelő-fák alkalmazásakor is felírhatók a pont és a terület lekérdezés algoritmusai:

- **Pont lekérdezés**: A lekérdezés lépései a következők lesznek:
  (1) A *P* pont címkéjének (*l*) meghatározása: a címkét úgy állítjuk elő, hogy a teret egy griddel fedjük le, amely olyan módon áll elő, mintha a negyedelés teljes lenne. Ekkor a *P* pontot tartozó cella címkéje a Z-vonal mentén és a negyedelés mélységének megfelelően meghatározható (ilyenkor a címke hossza egyenlő a fa mélységével).
  (2) A negyedelő fa levelének visszaállítása a B^+-fából: ha *L* annak a levélnek a címkéje a négyfában, amelyben a *P* pont szerepel, akkor két eset állhat elő:
    - *L* = *l*: a levél címkéjének hossza egyenlő a fa mélységével.
    - *L* prefixe *l*-nek: a levél címkéjének hossza kisebb a fa mélységénél és a levélhez tartozó negyed tartalmazza azt a cellát, amelyben a *P* pont található. Pontosabban az *L* a legnagyobb címke a B^+-fában, amely kisebb vagy egyenlő, mint *l*, és a négyfában a hozzá tartozó negyed tartalmazza a pontot.
  (3) Levél beolvasása és keresése: beolvassuk a levélhez tartozó blokkot és az összes bejegyzésre megvizsgáljuk a tartalmazást.

A fenti jelölések bevezetése után az algoritmus a következő [1]:

LQ-POINTQUERY(*P* : Point) : Set(*oid*)

```
Begin
  res := 0
  l := POINTLABEL(P)
  [L, p] := MAXINF(l)
  page := READPAGE(p)
  for each e in page do
    if(P in e.mbb) then res += {e.oid} end if
  end for
  return res
End
```

Az algoritmus tehát azon objektumok azonosítóit adja vissza egy listában, amelyek legkisebb befoglaló téglalapja tartalmazza a *P* pontot. A

POINTLABEL(*P*) függvény meghatározza az adott pont címkéjét. A *MAXINF*(*l*) függvény megkeresi a B^+-fában a (2) pontban leírtaknak megfelelő címkét. A *READPAGE*(*p*) beolvassa a *p* blokk tartalmát a memóriába. Mivel a komplex objektumoknak csak a legkisebb befoglaló téglalapjával dolgoztunk, azért a műveletek elvégzése után egy további finomításra is szükség van, amely megmondja, hogy maga az objektum valóban tartalmazza-e a keresett pontot. A lemez I/O műveletek száma a következő két részből tevődik össze: első lépésben a levelekhez vezető út során a belső csúcsokhoz tartozó blokkokat beolvassuk majd a megtalált levelet is beolvassuk. Ehhez összesen $d + 1$ lemez I/O műveletre van szükség, ahol *d* a B^+-fa magassága. A B^+-fa helyfoglalása és teljesítménye csak a fa alakjától függ, ezért az I/O műveletek számát nem befolyásolja a pontok térben való eloszlásának egyenletessége sem pedig a dinamikus szituációk. Még a legrosszabb esetekben is hatékony a működés.

- **Terület / ablak lekérdezés**: A feladat az, hogy meghatározzuk azt az intervallumot, amelybe eső címkékhez tartozó negyedeknek van átfedésük a megadott *W* ablakkal. A végeredmény meghatározásához egy tartomány lekérdezést kell lefuttatnunk a B^+-fán. Ennek három lépése a következő:
  (1) A *W* ablak bal felső csúcsához tartozó címke (*l*) előállítása a pont lekérdezés algoritmusának (1) lépésében leírt módon. A kapott címke segítségével és a pont lekérdezés (2) lépésében megadott algoritmus felhasználásával megkeressük a B^+-fában az intervallum alsó határát (*L*). Ugyanezzel a módszerrel kiszámítjuk az ablak jobb alsó csúcsának a címkéjét (*l'*) és a hozzá tartozó *L'* címkét. Ez lesz az intervallum felső határa.
  (2) Az [*L*, *L'*] intervallumon belül maradva lefuttatunk a B^+-fában egy tartomány lekérdezést, amely megadja az összes olyan bejegyzéshez tartozó címkét, ami az intervallumon belül helyezkedik el.
  (3) Az előző pontban meghatározott bejegyzésekhez meghatározzuk az általuk címkézett tartományi negyedeket. Ha a kapott negyed átfedésben vagy metszésben van a *W* ablakkal, akkor beolvassuk a hozzá tartozó blokkot és az összes bejegyzésre megvizsgáljuk a tartalmazást.

  A kapott eredmény többszörösen tartalmazhatja ugyanazokat az elemeket, ezeket rendezzük, majd a duplikátumokat töröljük. Az algoritmus azon objektumok azonosítóit adja vissza egy listában, amelyeknek van közös (teljes tartalmazás vagy metszet) részek a *W* ablakkal.

A lekérdezés algoritmusa a következő [1]:

LQ-WINDOWQUERY(*W* : Rectangle) : Set(*oid*)

```
Begin
   res := 0
   l := POINTLABEL(W.NW); [L, p] := MAXINF(l)
   l' := POINTLABEL(W.SE); [L', p'] := MAXINF(l')
   Q := RANGEQUERY([L, L'])
   for each q in Q do
      if (QUADRANT(q.l) overlaps W) then
         page = READPAGE(q.p)
         for each e in page do
            if(W ∩ e.mbb ≠ Ø) then res += {e.oid} end if
         end for
      end if
   end for
   SORT(res); REMOVEDUPL(res)
   Return res
End
```

Az [*L*, *L'*] intervallumban sok olyan negyed címkéje is előfordulhat, amelyeknek nincs a *W* ablakkal közös részük. Az I/O műveletek száma így szükségtelenül nagyra nő. Ugyanakkor a költség csökkenthető, ha azokat a blokkokat, amelyekben szereplő negyedek nem fedik a területet, nem is vizsgájuk meg.

Terület lekérdezésnél kicsit összetettebb az I/O műveletek kiszámításának módja. Az első lépésben, az intervallum két végpontjának megtalálásához kétszer annyi művelet szükséges, mint a pontkereséskor, azaz 2**d* lemezműveletet hajtunk végre. A második lépésben eljutunk ahhoz a levélhez, amelyik az intervallum alsó határát jelképezi, ez *d* lemezműveletet eredményez. A lépések után a tartomány végéig sorban beolvassuk a levelekhez tartozó összeláncolt blokkokat. Így tehát a további lemezműveletek száma a tartomány méretétől függ.

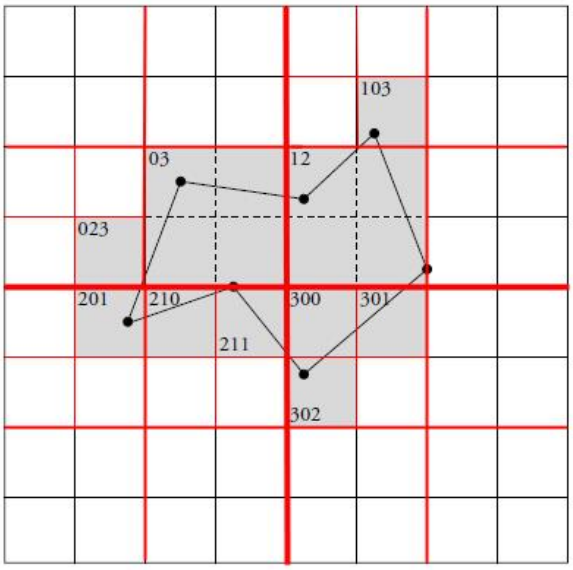
Léteznek sokkal hatékonyabb algoritmusok is, amelyek elkerülik azoknak a negyedeknek a keresését, amelyeknek a tartománnyal nincsenek közös részeik. De ezek nagyon megnehezítik a tervezés bonyolultságát.

4.9 Z-rendező fa

Ebben a fejezetben bemutatásra kerülő szerkezet az alakzatok közelítésére nem az *mbb*-t fogja használni. Jobban járunk, ha egy negyedelő-fa segítségével bontjuk őket fel, majd a levelekbe került térrészek segítségével adunk rájuk közelítést. Ismét egy négyfát építünk fel, majd a leveleinek címkéjét egy B^+-fába szervezzük.

A következőkben megadom a fa felépítésének rövid leírását. A gyökérből haladunk a fa levelei felé a következő módon [1]: A gyökér a teljes térrészt ábrázolja. Adott egy *o* objektum geometriája és egy *q* negyed. Ellenőrizzük, hogy az *o* lefedi-e a *q* gyökerű teljes négyfa leveleit. Ha lefedi, akkor a *q* eleme lesz az eredményhalmaznak. Ha az *o* nincs fedésben *q* valamely levelével, akkor *q* minden levelét további négy részre osztjuk. Az így létrejött új negyedekre ismét megvizsgáljuk a tartalmazást. A rekurzív negyedelést, addig végezhetjük, amíg el nem érjük a fa mélységére adott *d* korlátot. A fa egyes leveleiben létrejött negyedeket egyesítve, az *o* objektumnak egy legjobban közelítő lefedését kapjuk (4.23 ábra). Minden alkalommal egy negyed címkéje és a hozzá tartozó alakzat *oid*-je kerül be az eredményhalmazba a Z-vonalnak megfelelő sorrendben. A létrejött eredményhalmaz tekinthető az objektum raszteres közelítésének.

A negyedelő-fa felépítése után, ha vesszük a leveleit és egy B^+-fát szervezünk föléjük, akkor megkapjuk a Z-rendező fát. Ha a vonalban nagyobb ugrás van, akkor a létrejövő B^+-fában egy objektumot lefedő cellák egymástól távolra is kerülhetnek. Egyébként általában szomszédos levelekben lesznek.



4.23 ábra: Térfelbontás a Z-rendező fához

Ahogy a lineáris negyedelő-fáknál, itt is előfordulnak duplikátumok, mert egy objektum azonosítója az eredményhalmaznak annyi elemében fog szerepelni, ahány cellát az alakzat közelítése tartalmaz. Az is előfordulhat, hogy az eredményhalmaz több azonos címkével rendelkező eleméhez más és más *oid* tartozik. Ez az eset akkor áll fenn, ha egy cella, több alakzatnak a negyedekkel való közelítésében is szerepel és mindegyiknek egyforma mélységben. Az is előfordulhat, hogy egy cella több objektum közelítésében is részt vesz, de a felbontás különböző szintjén.

Ugyanúgy, ahogy a lineáris negyedelő-fáknál, itt is megadhatók a pont és a terület lekérdezés algoritmusai. A különbség az utolsó lépésben van, amikor az alakzat azonosítóját is eltároljuk, amely az éppen vizsgált aktuális cellával fedésben van.

A terület lekérdező algoritmus a következőképpen alakul [1]:

ZQ-WINDOWQUERY(*W* : Rectangle) : Set(*oid*)

Begin
  res := 0
  l := *POINTLABEL*(*W*.NW); [*L*, *p*] := MAXINF(*l*)
  l' := *POINTLABEL*(*W*.SE); [*L'*, *p'*] := MAXINF(*l'*)
  Q := *RANGEQUERY*([*L*, *L'*])
  for each *q* **in** *Q* **do**
    if (*QUADRANT*(*q*.*l*) overlaps *W*) **then** *res* += {*e.oid*} **end if**
  end for
  SORT(*res*); *REMOVEDUPL*(*res*)
  Return *res*
End

Terület lekérdezésnél az I/O műveletek száma ismét összetett módon történik. Az első lépésben, az intervallum két végpontjának megtalálásához kétszer annyi művelet szükséges, mint a pontkereséskor, azaz 2 * *d* lemezműveletet hajtunk végre. A második lépésben eljutunk ahhoz a levélhez, amelyik az intervallum alsó határát jelképezi, ez *d* lemezműveletet eredményez. A lépések után a tartomány végéig sorban beolvassuk a levelekhez tartozó összeláncolt blokkokat. Így tehát a további lemezműveletek száma a tartomány méretétől függ.

Összehasonlítva a lineáris negyedelő-fát és a Z-rendező fát, a következő megállapításukat tehetjük. Ahogy a struktúra nő, a B^+-fa mélysége és vele együtt az I/O

lemezműveletek száma is megemelkedik, mert minden objektum közelítésében résztvevő összes cella megjelenik egy bejegyzésben. Azt mondhatjuk, hogy az objektumok közelítése jobb lett, mint a lineáris négyfánál, de a bejegyzések száma drasztikusan növekedett. A lineáris negyedelő-fából készített B^+-fában a bejegyzések száma jóval kevesebb, mindig egyenlő a fa negyedeinek számával.

A Z-rendező fának vannak olyan variánsai is, amelyek a raszteres képek indexelésére jól használhatók. A módszer ugyanaz, mint a vektoros esetben, kivéve, hogy az eredményhalmazban szereplő térrészeket a legelemibb cellákra osztjuk fel. Egy objektumot így a legkisebb felbontási szinten tudjuk közelíteni. Ennek a típusnak nagy hátránya, hogy a redundancia elég nagy, és a bejegyzések száma jóval több, mint a vektoros ábrázoláskor. A pont és a terület lekérdező algoritmusok azonban egyszerűsödnek. Az ablak lekérdezések algoritmusánál nincs szükség a *MAXINF* függvényre az intervallum határainak meghatározására, mert minden cella minimális.

A bejegyzések számának csökkentéséhez létezik a Z-rendező fáknak egy redundancia nélküli típusa is, de a diplomamunkámnak nem célja a raszteres indexelési módszerek teljes körű bemutatása.

A területalapú módszereknél részletesen bemutattam a leggyakrabban használt indexelési módszereket. A következő fejezetben áttérek az adatvezérelt módszerekre.

5. Adatvezérelt módszerek

Az adatvezérelt módszerek nem a térből, hanem az objektumokból indulnak ki. Idesoroljuk azokat a hierarchikus szerkezeteket, amelyek a térbeli objektumokat szervezik össze a befoglaló téglalapjuk alapján. A térfelosztás helyett az objektumok pozícióját téglalapokkal közelítjük és így építjük fel a fákat. Ebben a fejezetben kap nagy szerepet az alakzatok legkisebb befoglaló téglalapjának fogalma.

5.1 R-fa

Térbeli objektumok indexelésére a legtipikusabb és legegyszerűbb adatvezérelt módszer az R-fa (*Region-tree*). Szerkezetük a térben elhelyezkedő téglalapok eloszlásától függ. A B-fákhoz hasonlóan, a szerkezetük hierarchikus felépítésű és kiegyensúlyozott. A különbség, hogy míg a B-fák egyszerű értékű kulcsokból épülnek fel, támaszkodva azok rendezettségére, addig az R-fák téglalapokat szerveznek össze.

Minden belső csúcs és a levelek a háttértároló egy-egy blokkjának felelnek meg, így a fa paramétereit úgy kell megválasztani, hogy a lekérdezések során minél kevesebb számú csúcselérés legyen, csökkentve ezzel az I/O műveletek számát. A belső csúcsokban egy téglalap szerepel, amely a csúcs egy blokkba eső gyerekeinek legkisebb befoglaló téglalapja lesz. A különböző csúcsok téglalapjai egymással lehetnek átfedésben is. A levelekben az adatbázis objektumaira találhatók hivatkozások. Bár a téglalapok a térbeli pozíciójuk szerint több levélhez is tartozhatnak, mégis a térfelosztó módszerekkel ellentétben most kizárólag egyhez fogjuk őket bejegyezni. Ez azt jelenti, hogy egy lekérdezés esetén több csúcsot is meg kell vizsgálni ahhoz, hogy megállapítsuk egy téglalap létezését vagy nem létezését. Magukat az objektumokat is az *mbb*-jükkel azonosítjuk (5.1 ábra).

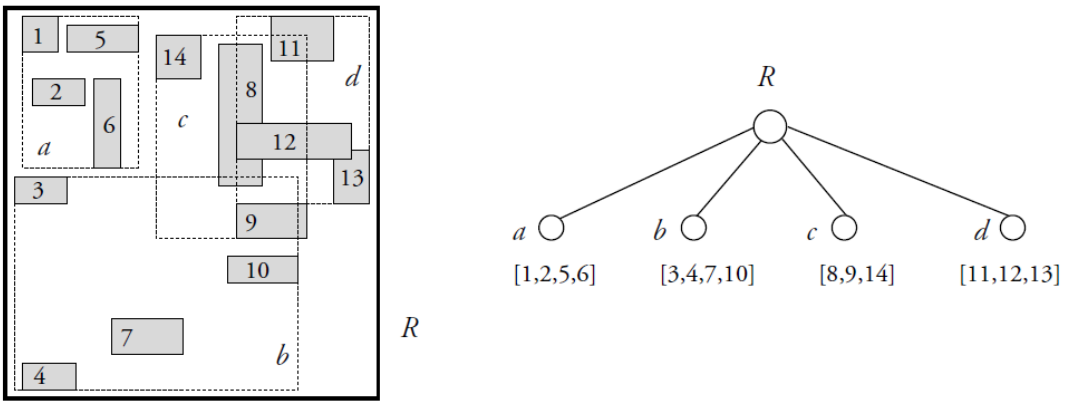
Ha a blokkok mérete *M*, akkor a fa paraméterezése a következő módon alakul [1]:

- minden csúcsban a bejegyzések száma *m* és *M* között van ($0 \le m \le \lceil M / 2 \rceil$).
- **belső csúcsok**: egy *N* belső csúcsnak (*dr*, *nodeid*) párok az elemei, ahol *dr* a legkisebb befoglaló téglalapja *N* azon gyerekeinek, amelyek a *nodeid* című blokkban szerepelnek.
- **levelek**: egy levélnek (*mbb*, *oid*) párok az elemei, ahol *mbb* a legkisebb befoglaló téglalap, *oid* pedig annak az objektumnak az azonosítója, amelyet a

téglalap tartalmaz. A befoglaló téglalap koordinátái: $(x_1, y_1, \ldots, x_d, y_d)$, ahol d a tér dimenziója.

- a gyökérnek legalább két eleme van, kivéve, ha levélcsúcs.
- a fa kiegyensúlyozott, levelei egy szinten vannak. A fa magassága n számú objektum indexelésekor legfeljebb $\lfloor \log_m n \rfloor - 1$ és legalább $\lfloor \log_M n \rfloor - 1$ lehet. A pontos érték a blokkok kihasználtságától függ.

Az R-fa alakja nem egyedi, szerkezete erősen függ a beszúrt téglalapok sorrendjétől. Az említett tulajdonságoknak és a fa kiegyensúlyozottságának minden új elem beszúrása után és egy elem törlése után is fenn kell állnia.



5.1 ábra: R-fa

Nagyszámú objektumok esetén a fa szomszédos leveleinek száma megnő, nem úgy, mint pl. a négyfáknál.

Említettem, hogy M a maximuma az egy csúcsban elhelyezkedő bejegyzések számának. Értéke függ az elemek méretétől és a blokk méretétől, azaz $M = \lceil SIZE(P) / SIZE(E) \rceil$ képlettel adható meg. Értéke, levélcsúcs és belső csúcs esetén különbözhet, mert a bejegyzések mérete az *oid* és a *nodeid* hosszától függ. Az *oid* nagyobb lesz, mert egy lapon belüli objektum címe biztosan hosszabb, mint egy lapcím (*nodeid*). A csúcsok minimális bejegyzéseinek száma m, amely a csúcsvágások stratégiáitól függ. Adott m és M paraméterű, d magasságú R-fa legalább $m^d + 1$ és legfeljebb $M^d + 1$ objektum indexelésére alkalmas.

Keresés R-fában [1]: A terület lekérdezés módszere, a pontlekérdezések általánosítása, ezért az utóbbit mutatom be részletesen. A pontlekérdezés a B-fáknál ismert algoritmushoz hasonlóan történik. A gyökérből indulva, annak minden gyerekét

megvizsgáljuk, hogy valamely hozzájuk tartozó *dr* téglalap tartalmazza-e a keresett pontot. Itt jegyzem meg, hogy az R-fa nem garantálja, hogy a megoldáshoz egyetlen utat kell csak bejárni a fában, mert az egy szinten lévő téglalapok lehetnek egymással fedésben is. Ha egy keresett pont vagy objektum több téglalap metszetébe esik, akkor a keresést ezeknek a részfáknak mindegyikére szükséges kiterjeszteni. Ez a tulajdonsága is megkülönbözteti az R-fákat a térfelosztó módszerektől. Legrosszabb esetben a gyökérből indulva az összes levélig vezető utat be kell járni így az indexhez tartozó minden blokkot be kell olvasni, ami a hatékonyságot csökkenti. A kereső algoritmust addig folytatjuk, amíg a fában egy levélhez nem érkezünk. Ha több részfában is keresünk, akkor mindegyikben egy levelet el kell érni. A keresés alatt az egyes csúcsok elérésekor két eset állhat elő:

- Az éppen elért csúcsban egyik bejegyzett téglalap sem tartalmazza a *P* pontot. Ekkor a keresés leáll. Ez akkor fordulhat elő, ha a *P* pont a csúcs szülőjében található *dr* téglalapjába beleesik ugyan, de az abban szereplő tartományok egyikébe sem, azaz a pont a csúcs holt-terében van.
- Az éppen elért csúcs egy vagy több bejegyzésében is szerepel a *P* pont. Ekkor minden tartalmazás esetén, az azokhoz tartozó részfát meg kell vizsgálni.

Második lépésben, ha egy levelet elértünk, akkor az abban szereplő összes bejegyzés címet meg kell vizsgálnunk, hogy pontosan melyik vagy mely objektumok tartalmazzák a keresett *P* pontot.

A fent leírtaknak megfelelő pontlekérdező algoritmus a következőképpen alakul [1]:

RT-POINTQUERY(*P* : Point) : Set(*oid*)

```
Begin
   res := 0
   SL := RTREETRAVELSAL(root, P)
   for each l in SL do
      for each e in L do
         if(P in e.mbb) then res += {e.oid} end if
      end for
   end for
   return res
End
```

Az algoritmushoz tartozó *RTREETRAVELSAL* rekurzív eljárás [1]:

```
RTREETRAVERSAL(nodeid : PageID, P : Point) : set of leaves
Begin
    res := 0
    page = READPAGE(nodeid)
    if (N is a leaf) return {N}
    else
        for each e in N do
            if(P in e.dr) then res += RTREETRAVELSAL(e.nodeid, P)end if
        end for
    end if
    return res
End
```

Ha a *P* pont minden szinten egyetlen téglalapba esik, akkor az algoritmushoz *d* blokkolvasás szükséges, ahol *d* a fa mélysége. Ez az eset nagyon ritka és ilyekor a keresések alkalmával csak néhány utat kell végigjárni a fában a gyökértől a levelekig. A műveletigény logaritmikus lesz. A legtöbb esetben azonban nem ez a helyzet áll elő.

A területlekérdezés algoritmusa a fentiekből általánosítható. A „*P in*” predikátum helyett az „*intersect W*” predikátumot használjuk, ahol *W* az ablaklekérdezés argumentuma. Minél nagyobb az ablak mérete, annál több csúcsot kell bejárni kereséskor.

Beszúrás R-fába [1]: Az új pontok illetve az objektumokat befoglaló téglalapok mindig a levelekbe kerülnek beszúrásra. A gyökértől indulva a levelek felé haladunk és minden csúcsban döntünk a továbbhaladás irányáról. A döntés feltétele az, hogy az éppen elért csúcsban melyik bejegyzéshez tartozó *dr* mérete alkalmas arra, hogy a beszúrandó objektumot magába foglalja. Ha egyiknek sem elég nagy a mérete, akkor olyat választunk, amelyet a legkevésbé kell megnövelni. Ha több ilyen is van, akkor a legkisebb területűt választjuk ki. (*CHOOSESUBTREE*) Egy *dr* növelése után, ellenőrizni kell, hogy a szülőt nem érinti-e a téglalap méretének növekedése (*ADJUSTPATH*). Legrosszabb esetben a gyökérig fel kell menni és minden szinten szükséges a méreteket frissíteni. Ezzel a módszerrel addig haladunk lefelé a fában, amíg egy levelet el nem érünk. Ott ellenőrizzük, hogy a hozzá tartozó blokk nem fog-e túlcsordulni a beszúrás

végrehajtásakor. Ha nem lép fel túlcsordulás, akkor a levélbe bekerül egy új (*mbb, oid*) pár, ellenkező esetben csúcsvágást kell végezni, majd az így létrejött új levelek között az objektumokat elosztani. Csúcsvágáskor ismét felfelé haladva ellenőrizni kell, hogy szükséges-e bármely belső csúcsot is kettébontani, hogy a fa alapvető tulajdonságai megmaradjanak és a lapokban ne legyenek túlcsordulások. (*SPLITANDADJUST*) Legrosszabb esetben a vágások sorozata a gyökérig felgyűrűzik. Ha a gyökérben is vágásra van szükség, akkor a fa mélysége 1-el meg fog nőni.

A belső függvényeket nem részletezve a beszúrás algoritmusa a következő [1]:

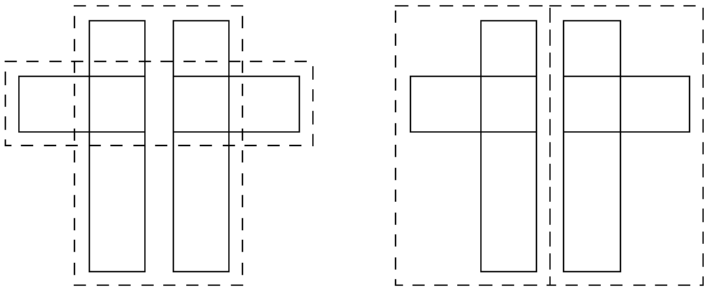
```
INSERT(e : LeafEntry)
Begin
    node := root
    while (node is not leaf) do
        node := CHOOSESUBTREE(node, e)
    end while
    INSERTINLEAF(node, e)
    if (node owerflows) then
        SPLITANDADJUST(node)
    else
        ADJUSTPATH(node)
    end if
End
```

Egy blokk túlcsordulásakor felmerülő csúcsvágáskor az objektumok szétosztására több megoldás is adódik. Figyelembe véve, hogy egy lapra minimálisan *m* bejegyzésnek kell kerülnie, a lehetséges megoldások a következők: egyik csúcsba $m + i$ bejegyzést, a másikba $M + 1 - m - i$ db bejegyzést szúrunk be, ahol $0 \le i \le M - 2m - 1$. Egy jól meghatározott csúcsvágásnak két alapvető feltételt kell kielégítenie (5.2 ábra):

(1) minimalizálja a létrejött két csúcs összes területét

(2) minimalizálja a létrejött két csúcs átfedésének nagyságát

Meg kell jegyezni, hogy két olyan erős feltétel van megfogalmazva, amelyek együttesen csak ritkán teljesülnek. Általában minimális terület esetén az átfedés nagy is lehet és fordítva. *Guttman* (1984) három olyan csúcsvágó algoritmust is adott, amelyek az (1) feltételen alapulnak [2].



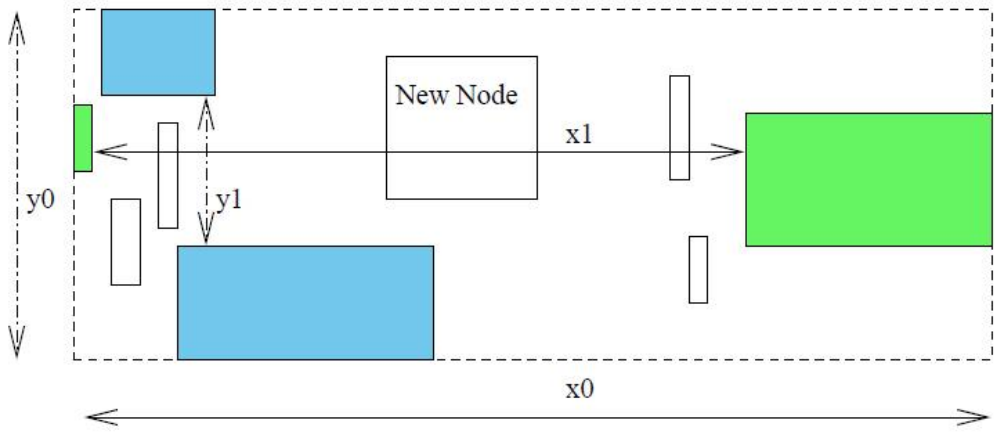
5.2 ábra: Csúcsvágás feltételei

Az első, egy nem túl nagy ötletet igénylő, kimerítő algoritmus, mert az összes lehetséges szétosztási lehetőséget és úgy választja ki az optimálisat. Az így létrehozható összes partícionálási lehetőségek száma $2^N - 1$ ($N + 1$ rekord szétosztásakor). Nagy N (pl.: $N > 50$) esetén ez a módszer ésszerűtlen. Az algoritmus költsége N növelésével exponenciálisan nő.

A második egy négyzetes költségű megoldás. Első lépésben megkeressük azt a két téglalapot, amelyek a legtávolabb vannak egymástól, azaz a holt tér maximális lesz. Ezt úgy kapjuk, hogy a szétvágandó lefedő téglalap területéből levonjuk a két téglalap területének az összegét. Tegyük ezt a két téglalapot külön csúcsba, nevezzük őket j-nek és k-nak. Ezt követően számítsuk ki az összes hátralévő téglalapra (i) d_{ij}-t és d_{ik}-t. d_{ij} a növekedés mértéke j-ben, ha az i téglalapot j-hez vettük hozzá. d_{ik}, hasonlóan d_{ij}-hez, a k növekedése lesz. A kiszámolt értékek alapján keressük meg a téglalapok közül azt (n), amelyre $| d_{ij} - d_{ik} |$ maximális, majd adjuk hozzá ahhoz a csúcshoz, amelyben a legkisebb területnövekedést okozza. Folytassuk ezt az eljárást, az összes hátralévő téglalapra. Az algoritmus költsége $O(M^2)$, ahol M az egy csúcsban maximálisan tárolható elemek száma.

A harmadik egy lineáris idejű eljárás. Az egyszerűbb leíráshoz az alábbi jelölést vezetem be: egy téglalapot a bal alsó és a jobb felső sarkának koordinátájával azonosítjuk, azaz $R = (L, B, R, T)$. Minden dimenzió mentén először megkeressük azt a téglalapot, amelynek az alsó oldala a legmagasabban van (R_1), majd megkeressük azt, amelynek a felső oldala a legalacsonyabban van (R_2). Pl. a síkban így 4 befoglaló alakzatot fogunk kapni. A téglalapok meghatározása után következik annak eldöntése, hogy a vágás melyik dimenzió mentén történjen, azaz az előbbiekben meghatározott téglalapok közül melyik kettő legyen a kiindulási alakzat. Ennek eldöntése, továbbra is a síkban maradva, a következő módon történik: az x tengely esetén meghatározott két

téglalap távolsága $y_1 = R_{1x}.T - R_{2x}.B$. Az y tengely esetén $x_1 = R_{1y}.R - R_{1y}.L$. A kapott két értékkel a szétvágandó csúcsban lévő téglalap oldalhosszait (x_0 és y_0) normálva a következő értékeket kapjuk: x_0 / x_1 és y_0 / y_1. Amely tengelyre ez az érték nagyobb lesz, azon dimenzió mentén hajtjuk végre a vágást. Az 5.3 ábrán az egyes tengelyek mentén talált két-két téglalap látható. Látható, hogy az y tengely fölötti két zöld téglalapra lesz a normalizált érték a nagyobb, így az *x* tengelyre merőlegesen fogunk vágni. A vágás után a maradék téglalapokat tetszőleges sorrendben vesszük és, ahhoz a csúcshoz soroljuk be, ahol az összterületet a legkevésbé fogják növelni. Az algoritmus költsége $O(M)$, ahol *M* az egy csúcsban maximálisan tárolható elemek száma.



5.3 ábra: Lineáris költségű csúcsvágás

Törlés R-fából [1]: A törlés műveletének tipikusan három lépése van, ahogy azt a fa struktúráknál már korábban is megszoktuk:

(1) törlendő elemet tartalmazó levél megkeresése

(2) elem törlése

(3) szükség esetén a fa átszervezése

A törlendő bejegyzés keresése ugyanúgy működik, mint a lekérdezések esetén. A gyökérből indulunk és a megfelelő levélig kell eljutni. Ha megtaláltuk az elemet tartalmazó levelet, akkor töröljük. Az algoritmus bonyolultabb részét ismét a fa átszervezése adja. A fa átszervezésére akkor van szükség, ha törölt elem után a levélben tárolt bejegyzések száma a minimális megengedett *m* szint alá esik. Ekkor a legegyszerűbb módszer az, hogy töröljük az egész levelet és a megmaradt $m - 1$ bejegyzés egy ideiglenes pufferbe kerül, majd újra beszúrjuk a fába. Ez a beszúrás a

leírtaknak megfelelő módon történik és eredményeképpen lehet, hogy az egész fa újraszervezésére szükség van.

Az algoritmus a következőképpen alakul [1]:

DELETE(*e* : LeafEntry)

Begin
  L := FINDLEAF(*e*)
  Q := REORGANIZE(*L*, e)
  REINSERT(*Q*)
End

REORGANIZE(*node* : Node, *e* : Entry)

Begin
  Q : set of nodes, initially empty
  node := *node* – {*e*}
  if (*node* is not root) **then**
    if (|*node*| < *m*) **then**
      F := GETPARENT(*node*)
      Q := *Q* ∪ REORGANIZE(*F*, entry of *node* in *F*)
    else
      ADJUSTPATH(*node*)
    end if
  end if
  return *Q*
End

A *FINDLEAF*(*e*) függvény megkeresi a fában azt a levelet, amely az *e* bejegyzést tartalmazza. Ezt a bejegyzést akarjuk törölni. A *REORGANIZE* rekurzív függvény segítségével töröljük a bejegyzést, majd alulcsordulás esetén az egész levelet. Rekurzívan, a levéltől a gyökérig vezető úton megvizsgáljuk az összes érintett csúcsot is (*ADJUSTPATH*). Ezek közül szintén eltávolításra kerülnek azok, amelyekben az elem vagy az egész levél törlése alulcsordulást okozott. A függvény azoknak a csúcsoknak a halmazával tér vissza, amelyeket töröltünk és a bejegyzéseiket a fába újra beszúrni szükséges. Miután az összes törölt csúcs elemeit összegyűjtöttük újra beszúrjuk őket a

gyökértől indulva. A *Q* elemei lehetnek levélbeliek vagy belső csúcsbeliek is. Ezeket a megfelelő szintekre kell újra beszúrni. A törléskor alkalmazott újra beszúrások folyamata lehetővé teszi a téglalapok jobb csoportosítását, ezáltal egy optimálisabb R-fa előállítását. Ugyanakkor a minősége nagymértékben függ a beszúrt téglalapok sorrendjétől. A fa első létrehozása a véletlenszerűen elhelyezkedő téglalapokból gyakran eredményez fontos átfedéseket a csomópontok között. Sőt, egy csomóponthoz rendelt bejegyzés lehet, hogy mindvégig az adott csúcsban marad, pedig a későbbiekben lehetne jobban is elhelyezni.

A törlés menetéből azonnal észrevehető, hogy itt igencsak különbözik az R-fa és a B-fa. Az utóbbinál, ha egy elem törlésekor alulcsordulás következett be, akkor csúcsösszevonást alkalmaztunk. R-fák esetén ez a teljes csúcs törlésével jár és a bejegyzések újra beszúrásából.

5.2 R^+-fa

Az R-fáknál a csúcsokban megengedettek a tartalmazó téglalapok közötti átfedések. Ám a kereséseknél bonyodalmat okoznak, mert előfordulhat, hogy így nem egyetlen utat kell bejárni a gyökértől egy levélig. Ennek következménye, hogy a kereséseknél legtöbb esetben a logaritmikus műveletigény nem tartható. Azt mondhatjuk, hogy a bejárt csúcsok száma függ a hozzájuk tartozó téglalapok átfedésétől. Ezeknek az átfedéseknek a kiküszöbölésére több különböző R-fa struktúrát találtak ki. Az egyik ilyen módosított R-fa adatszerkezet az R^+-fa. Ez a típus a kd-B-fa egyik kiterjesztése.

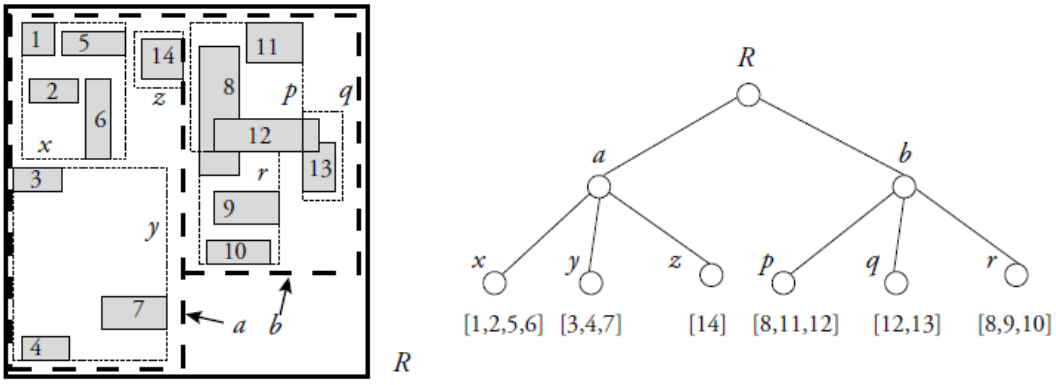
Az R^+-fák esetén az R-fákhoz képest annyi a módosulás, hogy a belső csúcsokban található *dr* téglalapok nem fedhetik át egymást. Ennek következménye, hogy egy pont lekérdező eljárás egyetlen utat fog követni a fában a gyökértől a megfelelő levélig, megkönnyítve ezzel a kereséseket. Az I/O műveletek számát így a fa mélységével felülről lehet becsülni.

A fentieknek megfelelően fa paraméterezése az R-fákhoz képest a következőképpen egészül ki [1]:

- a gyökérnek legalább két eleme van, kivéve, ha levélcsúcs.
- azonos szinten lévő *dr* téglalapok nem fedhetik egymást.
- **belső csúcsok**: egy *N* belső csúcs *dr* téglalapja befoglalja az *N* gyökerű részfa minden csúcsához tartozó téglalapokat.

- **levelek**: egy indexelt objektum befoglaló téglalapja minden olyan levélbe be fog kerülni, amelynek a *dr* téglalapjával valamilyen módon fedésben van (metszet vagy tartalmazás formájában). Ez azt jelenti, hogy egy objektum több levélhez is hozzátartozhat. Pl. 5.4 ábra *8*-as és *12*-es téglalapja.

A fa csomópontjainak szerkezete ugyanolyan lesz, mint az R-fáknál. Azonban, mivel nem garantálható a minimális tárolási kihasználtság *m* és mert a téglalapok duplikálódhatnak, ezért R^+-fa sokkal nagyobb lehet, mint egy ugyanazon objektumhalmazból felépített R-fa. Habár a duplikációk a fa magasságának növekedéséhez vezetnek, a keresések időben mégis felgyorsulnak. A fa felépítése és karbantartása a többi R-fa típushoz képest sokkal összetettebb.



5.4 ábra: R^+-fa

A keresés az R-fáknál látott módon történik. Pontkereséskor egyetlen utat szükséges csak bejárni. Téglalap kereséskor azonban a levelek szintjén duplikációkra kell számítani.

A következő részben röviden vázolom egy objektum dinamikus beillesztését a fába. Ezen algoritmus szerint tudjuk a fát felépíteni, illetve új téglalapokat beszúrni. Egy téglalap minden szinten azokba a csúcsokba kerül beszúrásba, amelyek *dr*-jával metszésben vannak. Ahhoz, hogy az egyes szinteken a *dr* téglalapoknál elérjük, hogy ne legyenek átfedésben egymással, ún. „nyírás” módszert alkalmazunk. Ha egy beszúrandó *R* téglalap több *dr*-t $\{r_1, \ldots, r_n\}$ is fed, akkor *R*-et kisebb téglalapokra osztjuk fel. A felosztás úgy történik, hogy minden létrejött rész egyetlen r_i-vel legyen csak metszésben. Ezt rekurzívan folytatjuk. A létrejött kisebb téglalapokat beszúrjuk az r_i gyökerű fába és a további darabolásokat és beszúrásokat addig alkalmazzuk, amíg el nem érünk egy levelet. Így állhat elő, hogy egy objektum több levélbe is bekerülhet,

attól függően, hogy a *mbb*-jét korábban hány részre kellett felosztanunk. A beszúráskor persze feltétel az is, hogy az r_i-k uniója, amelyekkel R metszésben van, R-et teljes egészében tartalmazzák. Ha nem így lenne, akkor az r_i-ket még növelni is kell. Ezt ugyanúgy tesszük, mint az R-fáknál a beszúrás algoritmusában (*ADJUSTPATH* függvény), kivéve, hogy a megnagyobbítás ne okozzon az r_i-k között átfedést és a fa R^+ tulajdonsága továbbra is fennálljon. Ez persze sajnos nem mindig lehetséges. Ilyenkor újra beszúrásra van szükség. Végül pedig, ha egy csúcs túlcsordul, akkor, ahogy az R-fánál is, itt is csúcsvágásra van szükség. Az algoritmus abban különbözik az R-fánál bemutatottaktól, hogy nyírást is kell alkalmazni. Ez az algoritmust bonyolítja, és a helykihasználtságot elronthatja.

Összességében a pont lekérdezések teljesítménye az R-fákhoz képest nőtt, kihasználva, hogy nincsenek átfedő téglalapjaink, így kevesebb számú csúcsot kell bejárni egy keresés során. Az ablaklekérdezések nyeresége már kevésbé nyilvánvaló. Az objektumok duplikációja nem csak növelte a fa magasságát, hanem potenciálisan drága utómunkához is vezetett (a duplikációk megszűntetéséhez rendezésre van szükség).

5.3 R*-fa

Az R-fák egy másik típusa az R*-fa. A mai térinformatikai adatbázisok többsége ezt a típust használja a tárolt objektumok indexelésére. A beszúrás algoritmusában vezettek be számos új fejlesztést, amelyek célja lényegében a következő paraméterek optimalizálása, azaz minimalizálása:

(1) csúcsok átfedése
(2) csúcsok által lefedett területek
(3) csúcshoz tartozó *dr* téglalap kerülete

Az utolsó pontbeli *dr* a minimális területű ilyen befoglaló téglalap. Szinte lehetetlen olyan optimalizáló technikát találni, amely mindhárom feltételt kielégítené. A következő részben két olyan variációt mutatok be, amelyek az R-fának a legjelentősebb fejlesztései.

Az első a csúcsvágó algoritmusnak egy fejlesztése. Az eddigiek során, ha egy csúcs kapacitása túlcsordult, akkor allokáltunk egy új lapot és a keletkezett két csomópont között osztottuk el az $M + 1$ darab bejegyzést. A megoldás tere tehát exponenciálisan függ M értékétől, ezért nem is lehet az egyetlen, legjobb eloszlást eredményező

megoldást megtalálni. Az R-fa vágóalgoritmusa először inicializálja a két csoportot meghatározó bejegyzéseket, amelyek a lehető legtávolabb vannak egymástól. Ezután hozzárendeli az egyes csoportokhoz a fennmaradó bejegyzéseket. Az R*-fa megközelítése olyan értelemben eltérő, hogy feltételezi, hogy a vágás az egyik tengely irányában történik és megvizsgálja az összes lehetséges elosztást, a vonalat felfelé illetve lefelé tolva. Az R*-fa tehát kiválasztja pl. az *x* tengelyt és a két csomópont között talál egy olyan partíciót, amelyeket a kiválasztott tengellyel párhuzamos egyenes választ szét és az átfedésük minimális. Annak érdekében, hogy megtaláljuk a legalkalmasabb tengelyt, minden egyes téglalap alsó és felső csúcspontját rendezni kell. Ennek számításigénye 2 * (2 * (M – 2 * m + 2)), algoritmusa pedig a következő [1]:

R*TREECHOOSEAXIS(E : Set of entries)

Begin
  for each axis a **do**
    sort E with respect to a
    $S := 0$
    for($i = 1; i < (M - 2 * m + 2); i$++) **do**
      $G_1 = E[1, m + i - 1]$; $G_2 = E[m + i, M + 1]$
      $Mg :=$ perimeter(mbb(G_1)) + perimeter(mbb(G_2))
      $S := S + Mg$
    end for
    if ($S <$ *min_perimeter*) **then**
      best_axis := a
      min_perimeter := S
    end if
  end for
  return *best_axis*
End

A módszer, a létrejött két téglalap kerületének összegét minimalizálja, így jut el a végső megoldásig.

A másik fejlesztés az R*-fa beszúró algoritmusában a kénytelen vissza-stratégia. Ahogy korábban, most is erőteljesen befolyásolja a bejegyzések beszúrási sorrendje a fa alakját és szerkezetét. Ahhoz, hogy a torzulásokat elkerülje, mindannyiszor, amikor túlcsordulás keletkezik, újraszervezést alkalmaz. A már eltárolt bejegyzések újra

beszúrásával elkerülhető a vágás. Az újra beszúrandó bejegyzésre azt választjuk ki, amely a *dr* téglalapban a legnagyobb holt teret hozza létre. Kizárólag arra kell figyelni, hogy arra a szintre szúrjuk be őket ismételten, ahol eredetileg is voltak, mert a túlcsordulás bármely szinten előfordulhat. Ezt viszont már az R-fa *DELETE* algoritmusa garantálja. Figyelni kell arra is, hogy az újra beszúrások miatt ne kerüljünk végtelen ciklusba. Ezért, ha az átrendezés alkalmával valamely csúcs ismét túlcsordul, akkor már vágást kell alkalmazni.

Az újra beszúrandó elemek kiválasztásának lépései a következőképpen alakulnak [1]:

(1) *dr* centroidja és minden benne lévő téglalap középpontja közötti *d* távolság kiszámítása.
(2) távolságok csökkenő sorrendbe rendezése.
(3) vesszük az első *p* értéket a rendezett sorból (*p* optimális esetben a sor hosszának 30%-a).
(4) a kiválasztott *p* elem újra beszúrása.

A javított vágó algoritmus és az egyes bejegyzések újra beszúró stratégiája egy sokkal jobban szervezett struktúrát ad az R-fából. Az adatszerkezet semmit nem változik, így az R-fán definiált adatkereső műveletek az R^*-fára is érvényben maradnak. Ám mivel az R^*-fa szervezettebb, ezért e műveletek is valószínű, hogy hatékonyabban fognak működni.

Az R-fánál, az R^+-fánál és az R*-fánál használt beszúró és törlő algoritmusok működése dinamikus, azaz egy már előre felépített fába is tudunk új bejegyzéseket beszúrni vagy régieket törölni. Ez olyan térinformatikai adatbázisok kezelésekor hasznos, amikor a tárolt adatok bizonyos rendszerességgel frissítésre szorulnak, illetve amikor új adatok felvitele válik szükségessé. Ezeknek a műveleteknek az egymás után többszöri alkalmazása azonban a helykihasználtság rovására megy. Statikus esetben, amikor már teljes egészében rendelkezésünkre áll az indexelésre váró adathalmaz, egy előfeldolgozás után optimálisabb fát tudunk építeni, úgynevezett pakolt algoritmusok segítségével. Ilyen algoritmus eredményeképpen kapjuk az R-fáknak egy újabb típusát, a pakolt R-fát. Ennek a szerkezetnek az ismertetését a statikus mivolta miatt nem teszem meg, csak megemlítettem, mert a térinformatikában fontos szerepet kaphat olyan adatbázisok kezelésekor, amelyek nem igényelnek állandó adatváltoztatásokat.

5.4 R-fák költség-modellje

A költség-modellek nagyon hasznosak egy művelet teljesítményének becsléséhez a térbeli adatbázis fizikai paramétereinek függvényében. Ilyen fizikai paraméterek lehetnek pl. indexek mérete, rekordok mérete, szelektív kiválasztási predikátumok.

A következő modell egyszerű becslést ad az R-fák műveletinek teljesítményére: Az egyszerűség miatt a keresési teret most egy egységnyi oldalú, normalizált négyzetnek vesszük, azaz legyen $[0, 1]^2$. Legyen N_i az *R*-rel jelölt R-fa egy csúcsa, amelyhez tartozik egy $N_i.dr$ téglalap ($s_{i,x}$, $s_{i,y}$) méretekkel. Tegyük fel, hogy pontkereső lekérdezéseket futtatunk az R-fán, melyeknek argumentumai egyenletes eloszlásúak. Annak valószínűsége, hogy a lekérdezés eredménye az N_i csomópontba esik:

$$PQ(N_i) = s_{i,x} \times s_{i,y}$$

A valószínűség tehát az $N_i.dr$ méretének és a keresési tér méretének (ami most 1) hányadosa lesz. A lekérdezés azokat a csúcspontokat érinti, amelyekben található *dr* téglalap tartalmazza a keresés argumentumában szereplő pontot. Az érintett csúcsok számát a következő módon lehet kifejezni, ahol *n* az R-fa csúcsainak száma:

$$PQ(R) = \sum_{i=1}^{n} PQ(N_i) = \sum_{i=1}^{n} s_{i,x} \times s_{i,y}$$

A formula kiterjeszthető az ablaklekérdezésekre is: Adott egy *N* csúcs és az ablaklekérdezés argumentuma *w*, melynek mérete (w_x, w_y). Becsüljük meg a valószínűségét annak, hogy *N.mbb* metszésben van *w*-vel. Ezt megtehetjük úgy, hogy visszavezetjük a pont esetére. A *w* ablakot a jobb-felső sarokpontjával azonosítjuk, és arra alkalmazzuk az előbbi – pontokra vonatkozó – valószínűséget. Erre két lehetőség adódik:

(1) az ablak jobb-felső sarka beleesik az *N* csúcshoz tartozó téglalapba.

(2) az ablak jobb-felső sarka beleesik az *N* csúcshoz tartozó téglalap „felfújtjába”

Ebből következik, hogy a *w* akkor és csak akkor metszi *N.mbb*-t, ha a jobb-felső csúcsa beleesik a $W = (N.mbb.x_{min} + w_x,\ N.mbb.y_{min} + w_y)$ téglalapba. Mivel feltételeztük, hogy az ablak argumentumok egyenletesen helyezkednek el a keresési térben, így a költség becsülhető úgy, mint egy pontlekérdezés műveletigénye *W* fölött, azaz:

$$WQ(w) = \sum_{i=1}^{n} (s_{i,x} + w_x) \times (s_{i,y} + w_y) =$$

$$= \sum_{i=1}^{n} s_{i,x} \times s_{i,y} + w_y \times \sum_{i=1}^{n} s_{i,x} + w_x \times \sum_{i=1}^{n} s_{i,y} + n \times w_x * w_y)$$

A szabály kizárólag az R-fa tulajdonságaitól függ és nem a fa típusától (R^+-fa, R^*-fa) és nem is a konstrukciójának módjától (statikus, dinamikus). Ezek a képletek erősen támaszkodnak arra a feltételezésre, hogy az adatok a eloszlása a térben egyenletes.

Az R-fákban való keresésekből, a rajtuk végzett beszúró és törlő algoritmusokból jól látható, hogy mindegyik esetében az átfedés, a tartalmazás illetve a metszet fogalma a középpontban állnak. Innen már sejthető, hogy a fáknak ezek a típusai elsősorban a tartalmazások és azzal kapcsolatos pont és ablak lekérdezésekre használhatók jól. Ebből adódóan szóba jön sokszor a téglalapok metszetének problémája is. Ezeknek az átfedéseknek további speciális változata, amikor két tábla térbeli feltétel alapján egy *join* művelettel össze van kapcsolva és a két objektumhalmaz elemei közül szeretnénk a térbeli kapcsolat alapján párokat kiválogatni. Ilyen térbeli kapcsolat lehet a lefedés, a metszet vagy a tartalmazás.

A későbbiekben hatékonyan alkalmazták őket egy pont (k) legközelebbi szomszéd problémájának megoldására is. Egy o objektum *mbb* téglalapja és egy P pont között kétféle távolságot is definiáltak, amelyek segítségével adható egy alsó illetve egy felső korlát a P pont és az o objektum távolságára.

A következő alfejezetben bemutatásra kerülő adatszerkezet és annak módosított változata nem sorolható szorosan az adatvezérelt indexelési módszerek körébe. A térinformatikai struktúrákkal foglalkozó könyvek legtöbbje azonban közös fejezetben tárgyalja őket az R-fákkal, mert a minimális befoglaló téglalapokkal kapcsolatos feladatok megoldására jól alkalmazhatók. Bizonyos térbeli kérdések megválaszolásának gyorsításához, akár indexeknek is felfoghatjuk őket.

5.5 Szegmensfa és intervallum fa

Ebben az alfejezetben a térinformatika újabb hasznos adatstruktúráját, a szegmensfát mutatom be. Az irodalomban több helyen ezt nevezik intervallum fának, ami azonban felületes fogalomalkotás, mivel ez utóbbi a szegmensfa egy hatékonyabb változata.

Lényegüket tekintve azonban alig különböznek egymástól. Mivel a szegmensfa jóval egyszerűbb adatszerkezet, ezért alapvetően csak ennek az alkalmazását ismertetem.

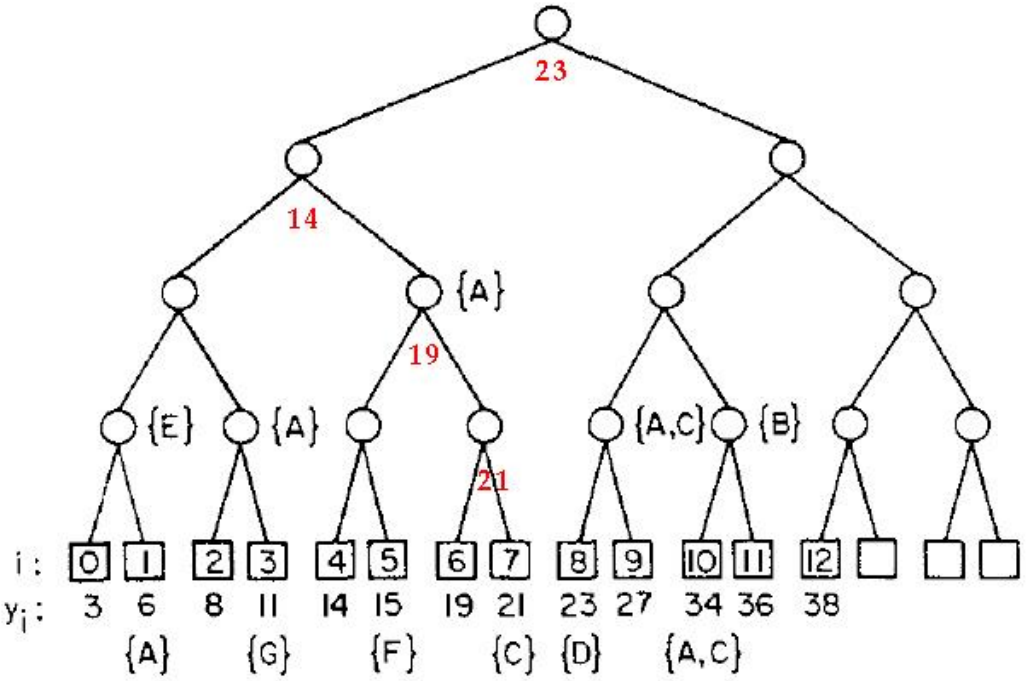
A szegmensfa a számítástudományban intervallumok és szegmensek tárolására szolgáló adatstruktúra, azaz olyan bináris keresőfa, amelynek elemei szegmensek lesznek. Jellemzően a térinformatika egyik gyakori síkbeli feladatának a megoldására alkalmas, a különböző tartalmazások és metszetek vizsgálatára használjuk. Segítségével hatékonyan tudunk arra a kérdésre válaszolni, hogy egy téglalapokból álló rendszerben melyek azok, amelyek egy megadott pontot tartalmaznak. Hasonló kérdés tehető fel a téglalapok átfedésével kapcsolatosan is: adott téglalaphalmazból melyek azok, amelyek metszik (átfedik vagy tartalmazzák) egymást. Ha téglalapokra valóban jól működik a struktúra, akkor a téglalapokból álló rendszer elemei tekinthetők síkbeli objektumok legkisebb befoglalóinak, így bizonyos finomítások után az objektumok átfedésének kérdése is megválaszolhatóvá válik. A szegmensfát használják még a számítógépes grafikában térbeli objektumok egy elrendezésében a láthatósági viszonyok meghatározására is, de ezzel az alkalmazási területükkel a dolgozatomban nem foglakozom.

5.5.1 A szegmensfa két változata

A szegmensfát először egy dimenzióban vezetem be, amikor alakzataink még nem téglalapok, hanem intervallumok. Két eltérő változatával találkozunk az irodalomban. Először a gyakoribb, de elméletileg kevésbé tiszta változatot ismertetem.

Első változat [2]: Adott n számú intervallum. Legyenek ezek pl. a síkban elhelyezkedő téglalapok vízszintes oldalszakaszai. Vegyük pl. a következő intervallumokat: $A = [6, 36]$, $B = [34, 38]$, $C = [21, 36]$, $D = [23, 27]$, $E = [3, 8]$, $F = [15, 19]$ és $G = [11, 14]$. Ekkor $n = 13$, mert a 36 két intervallum végpontjaként is előfordul. Ezt figyelembe véve megjegyzem, hogy ha pl. felvennénk még a $H = [11, 15]$ intervallumot is, akkor n értéke nem változna, mert H mindkét végpontja már szerepel a felvett pontok között. Ezek ismeretében tekintsünk egy olyan teljes bináris keresőfát, amelynek a leveleiben balról jobbra haladva az osztópontok növekvően rendezett sorozata található. A fa teljessége miatt további három üres levél is megjelenik (5.5 ábra). Az üres levelekre nem feltétlenül van szükség, ha a 12-es sorszámú, 38-as értékű levelet két szinttel feljebb visszük. Ha ezt megtesszük, a fa akkor is kiegyensúlyozott, tehát a legkisebb magasságú, amelyben ez a hét szakasz ábrázolható. (A teljességnek

minden esetben teljesülnie kell, tehát ha kilenc osztópont esetén hét üres levélre lenne szükség a fában, ami az ábrázolást és a helyfoglalást tekintve hátrányos lenne.)



5.5 ábra: szegmensfa első változata

Az ábrán a fa levelei egy-egy elemi intervallumot reprezentálnak, ezek sorban a következők: [3, 6], [6, 8], [8, 11], …, [36, 38], [38, ∞). Az utolsó levél – formálisan – egy jobbról végtelen intervallumot ábrázol, de felfogható egyetlen pontnak, az intervallumsorozat záró pontjának is.

A belső pontokban szintén intervallumokat jelenít meg: minden csúcshoz egy olyan szakasz tartozik, amely a gyerekei által reprezentált intervallumok uniója lesz. Pl. a 0 és 1 sorszámmal ellátott levelek szülőjében a [3, 8] intervallum található.

A szegmensfa bizonyos csúcspontjaiban az intervallumazonosítókat, mint címkéket láthatjuk, amelyeket egy listában tárolunk. Egy adott azonosító, pl. az *A*, azokat a csúcsokat címkézi, amelyek által reprezentált szakaszok uniója kiadja az *A* intervallumot. Az unióban szereplő szakaszok esetén feltétel, hogy a lehető legközelebb essenek a gyökérhez. Ez a címkézés tekinthető úgy is, mintha egy szakaszt (pl. *mbb* egyik oldalát) szegmensekre darabolnánk fel úgy, hogy a szegmensek mérete a lehető legnagyobb legyen. Pl. az *A* szakasz esetén a következő partíciókat kapjuk:

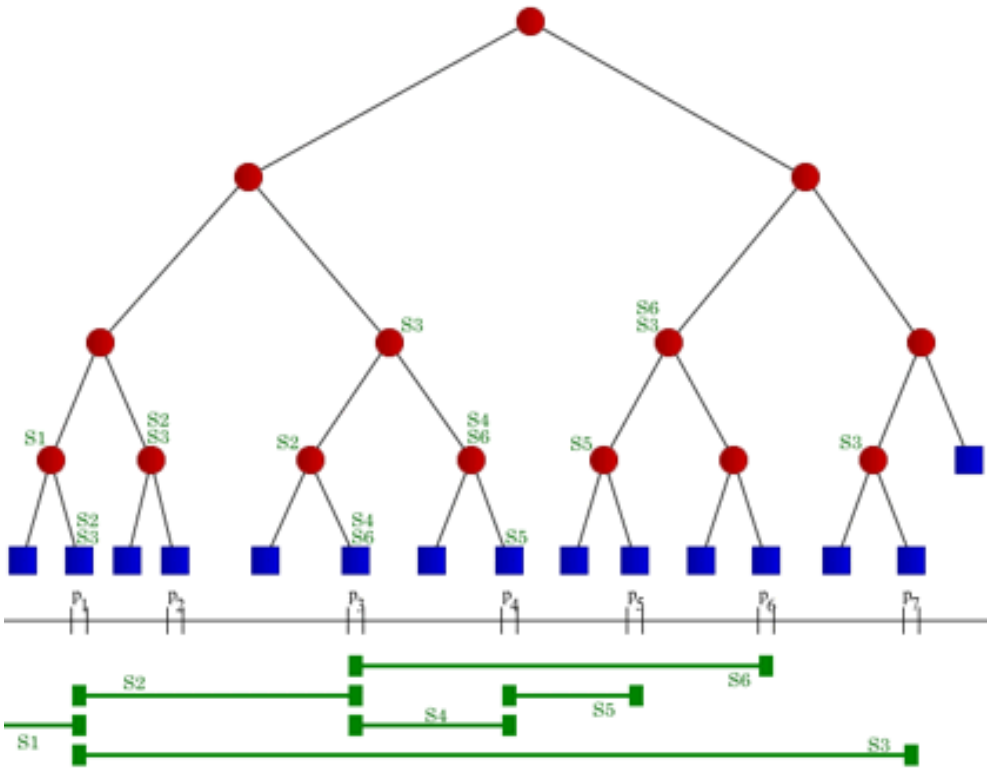
$$[6, 8] \cup [8, 14] \cup [14, 23] \cup [23, 34] \cup [34, 36] = [6, 36] = A$$

Könnyen belátható az a fontos tulajdonság, hogy egy intervallum a szegmensfa egy szintjén legfeljebb kétszer fordulhat elő.

Ennek a változatnak „gyengéje”, hogy az intervallumok zártak, így határpontjaik két érintkező elemi intervallumhoz is besorolhatók. Az F = [15, 19] intervallum az 5-ös számú levelet címkézi. Ha szerepelne a korábban említett H = [11, 15] szakasz, akkor az a 3-as és 4-es számú leveleket címkézné. Noha F-nek és H-nak közös pontja a 15, mégis, mint címkék elkülönülnek a fában. Erre a kétértelműségre a lekérdezések során majd tekintettel kell lenni.

Megjegyzem még, hogy a fa belső pontjaiba, a B-fához hasonlóan, a lekérdezéseket támogató kulcsértékeket helyezünk el. Minden csúcsba a belőle leágazó jobboldali részfa baloldali levelében szereplő kezdőértéket írjuk. (A 3.6 ábrán egy ilyen útvonal van feltűntetve.)

Második változat: Az előző verziótól csak az intervallumok osztópontjainak a kezelésében különbözik (5.6 ábra). Ebben az elméletileg tisztább változatban az adatstruktúra helyfoglalása nagyobb, de a rajta dolgozó algoritmusok egyszerűbbek és jobban áttekinthetők.

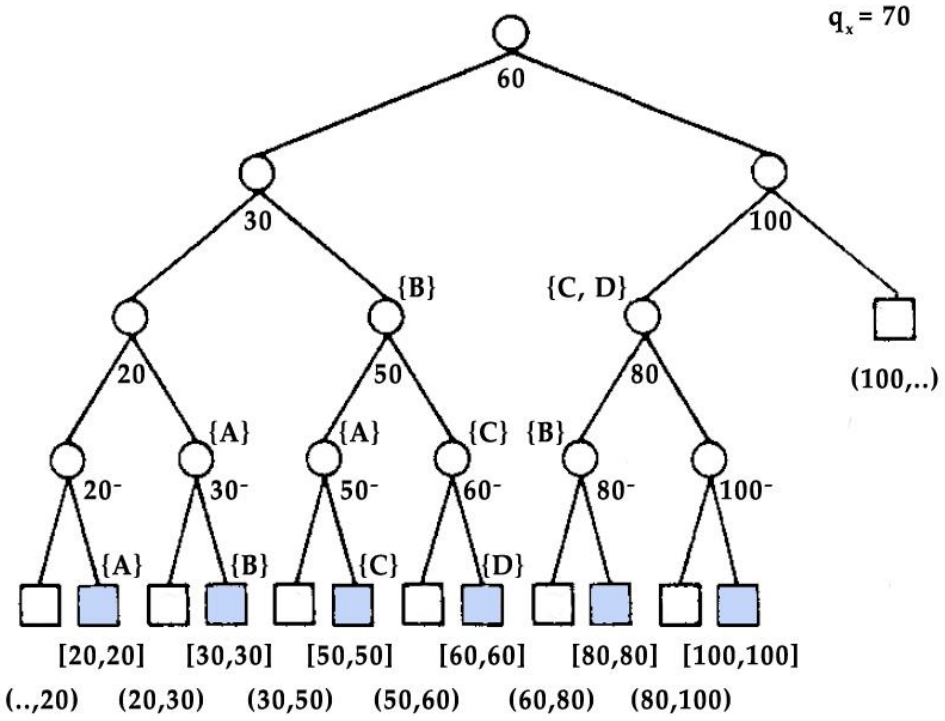


5.6 ábra: szegmensfa második változata

Ha *x* és *y* két egymás melletti osztópont, akkor belőlük három intervallum képződik: [*x*, *x*] pont, az (*x*, *y*) nyílt intervallum és az [*y*, *y*] pont. Az első ponttól balra és az utolsó ponttól jobbra két végtelen intervallum képződik.

Ebben a változatban az *n* intervallum legfeljebb 2*n* osztópontja legfeljebb 4*n* + 1 elemi intervallumot határoz meg, így a szegmensfa helyfoglalása nagyobb, mint az előző változatban, de nagyságrendben nem haladja meg azt és a rajta futtatott keresések is egyszerűbbé válnak.

Tekintsük a következő példát, amelyben négy intervallum szerepel: *A*[20, 50], *B*[30, 80], *C*[50, 100] és *D*[60, 100]. Az egybeesések miatt a következő 6 osztópontot kapjuk: 20, 30, 50, 60, 80 és 100. Így, összesen 13 elemi intervallum alakul ki: 6 pontszerű, közöttük 5 véges intervallum és a két szélén két végtelen tartomány. Ennek megfelelően felépítünk egy olyan kiegyensúlyozott bináris keresőfát, amelynek a 13 levele az elemi intervallumokat jelöli (5.7 ábra).



5.7 ábra: példa szegmensfa második változatára

A belső pontokban ismét intervallumokat jelenít meg: minden csúcshoz egy olyan szakasz tartozik, amely a gyerekei által reprezentált intervallumok uniója lesz. A keresőfa belső csúcsaiban a keresést támogató kulcsértékek kerülnek, amelyek mindig a baloldali részfa maximális kulcsértékével egyeznek meg. Megjegyzem, hogy

közvetlenül a levelek szintje fölött speciális kulcsértékek állnak, pl. a 60^- egy olyan értéket jelöl, amelynél a 60 nagyobb, de bármely 60-nál kicsit kisebb szám (pl. 5,99…) már kisebb.

Az intervallumok azonosítóit, az első verziónál leírt módon, címkeként egy listában elhelyezzük a fa megfelelő csúcsaihoz. Ha megnézzük pl. a *B* intervallumot, annak címkéje három csúcsnál szerepel, amelyek által reprezentált intervallumok uniója kiadja a *B* intervallumot:

$$B = [30, 80] = [30, 30] \cup (30, 60] \cup (60, 80]$$

Az unióban szereplő szakaszok esetén feltétel, hogy a lehető legközelebb essenek a gyökérhez.

Könnyen belátható ebben az esetben is az a fontos tulajdonság, hogy egy intervallum a szegmensfa egy szintjén legfeljebb kétszer fordulhat elő.

Helyfoglalás: A szegmensfa adatszerkezet helyfoglalása mindkét változat esetén $O(n * \log n)$, ahol *n* továbbra is az intervallumok száma. Az elemi intervallumok száma az első esetben legfeljebb $2n$, a másodikban legfeljebb $4n + 1$. Az ezekből felépített kiegyensúlyozott fa magassága mindkét esetben $O(\log n)$. Figyelembe véve azt, hogy minden intervallum a fa egy adott szintjén legfeljebb két csúcshoz tartozó listában szerepel, kapjuk az $O(n * \log n)$ helyfoglalást.

Konstrukció: A fa felépítését abban a statikus szerkezetben tekintem át, amikor az intervallumok halmaza ismert (már mind beolvasásra kerültek). A felépítés pontokba foglalt eljárása a következő:

(1) Az intervallumok végpontjait rendezéssel sorba állítjuk. Ennek műveletigénye $\Theta(n * \log n)$.

(2) Eltérően a korábban megismert struktúráktól, most a levelekből indulunk ki és az elemi intervallumok fölé egy kiegyensúlyozott fát emelünk. Ennek műveletigénye $O(n)$, mert egy kiegyensúlyozott fában közel ugyanannyi belső csúcs van, mint ahány levél. Eközben a belső pontokhoz az unió művelet segítségével meghatározzuk az általuk reprezentált intervallumokat.

(3) A gyökértől indulva a levelek felé, minden egyes adott intervallumot címkeként elhelyezünk a megfelelő csúcspontokhoz tartozó láncolt listában. Mivel minden intervallum a fa egy adott szintjén legfeljebb kétszer szerepel, ezért az összes intervallum elhelyezése $O(n * \log n)$ lépésben megtehető.

A szegmensfa létrehozásának teljes műveletigénye a fent leírtak szerint $O(n * \log n)$ lesz.

Intervallum beszúrása, illetve törlése: A beszúrás és a törlés műveletével a szegmensfa esetén nem foglalkoztam, mert többnyire olyan térinformatikai adatbázisok esetén optimális a használatuk, amelyeknél ritkán van szükség adatmódosításokra. A statikus szerkezet és a bonyolult beszúrás és törlés művelete miatt mondható ez. A két művelet rendre elvégezhető $O(\log n)$ és $O(n * \log n)$ műveletigénnyel.

5.5.2 Pont tartalmazás vizsgálata

A következő részben a szegmensfa mindkét típusára bemutatom az algoritmust, hogy hogyan ellenőrizhető a pont tartalmazása valamely intervallumra.

Első változat esete: A keresés minden esetben egy vagy két levél eléréséig tart, a keresés argumentumában szereplő pont tulajdonságától függően. Az outputban azoknak a címkéknek a listáját kapjuk, amelyekhez tartozó intervallumok tartalmazzák a keresett pontot.

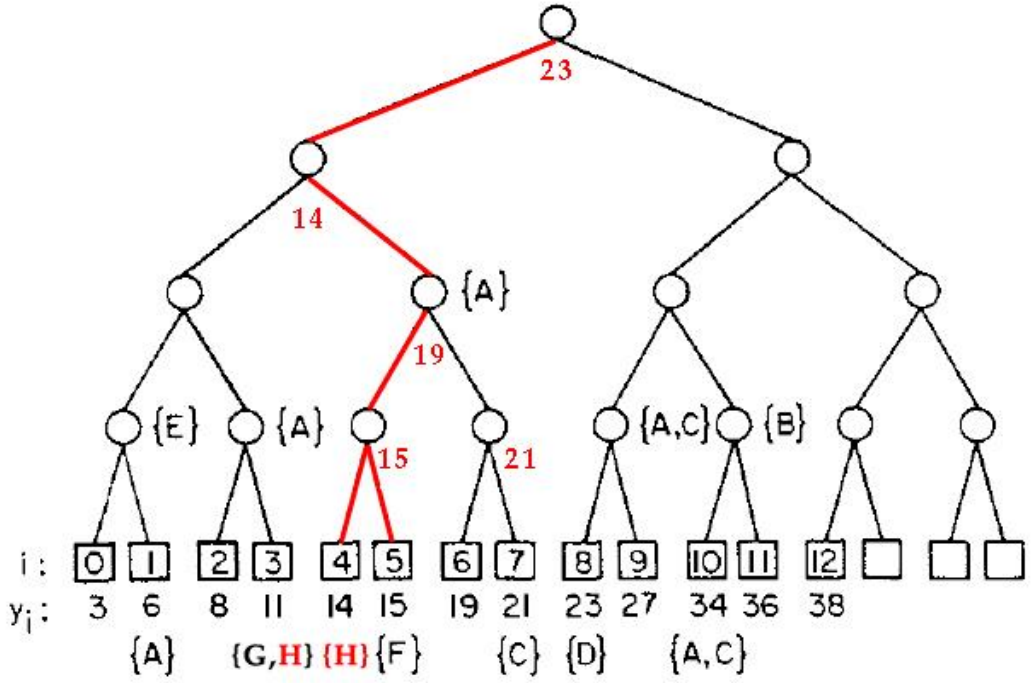
Pont tartalmazás vizsgálatakor a gyökérből indulunk és minden szinten az adott csúcsban elhelyezett, keresést támogató kulcsértékkel hasonlítjuk össze a pontot reprezentáló értéket. Ennek megfelelően döntünk a továbbhaladás irányáról. Ha egy olyan csúcsba érünk, amelynél szerepel egy címke, akkor a címkét az output listába felvesszük, mert a hozzá tartozó intervallum biztosan tartalmazni fogja a pontunkat.

Ha olyan pontot vizsgálunk, amely egyik intervallumnak sem kezdőpontja vagy végpontja, azaz, nem szerepel keresési kulcsként egyetlen csúcsban sem, akkor a lekérdezés útvonala a fában egyértelmű. Ilyenkor egyetlen utat járunk be a gyökértől valamely levélig. Ha a tartalmazás vizsgálatához olyan pontot használunk, amely megegyezik valamely osztóponttal, akkor a keresési út annál a csúcsnál kettéágazik, amelyben ez az érték szerepel keresési kulcsként. Ennek oka az, hogy egy osztópont lehet egyszerre egy adott intervallum végpontja és egy másik kezdőpontja is. Az előbbi, ha létezik, akkor az elágazás csúcsától balra helyezkedik el, az utóbbi, ha szintén létezik, akkor az elágazás pontjától jobbra helyezkedik el.

Az utak bejárása során, az output listába gyűjtött címkékkel jelölt intervallumok mindegyike biztosan tartalmazni fogja a lekérdezés argumentumában szereplő pontot.

Tegyük fel, hogy a korábbi szakaszban említett $H = [11, 15]$ intervallumot is felvesszük a fába (5.8 ábra) és keressük a 15-ös értéket tartalmazó intervallumokat. Az

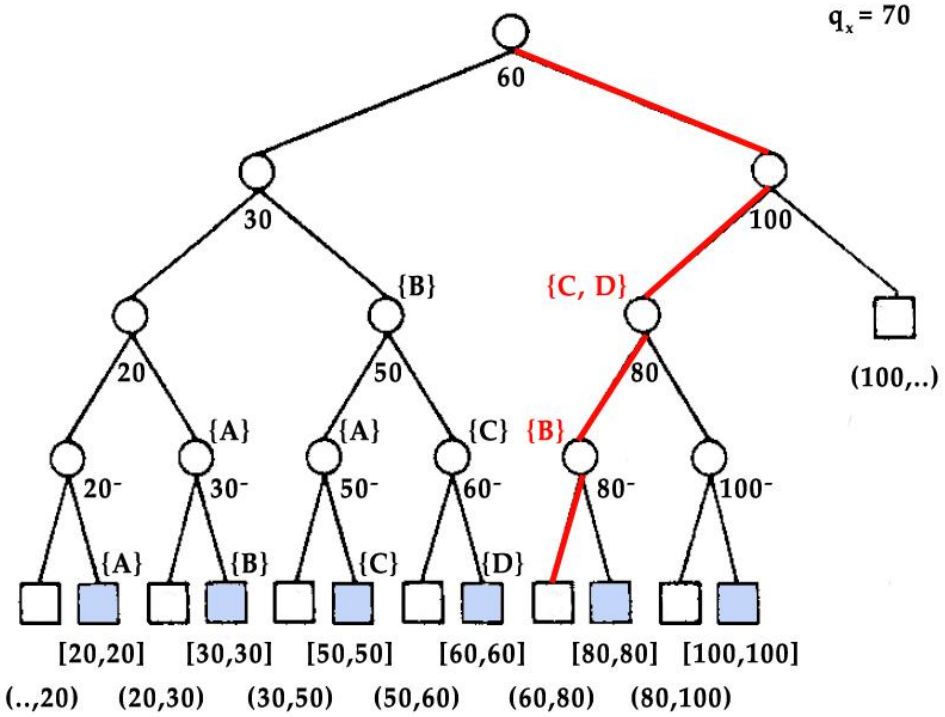
ábrán pirossal jelölt útvonalon haladunk lefelé a fában, minden csúcsban a 15-ös értéket az adott keresési kulccsal összehasonlítva. Mivel ez az érték egy osztópont, ezért egy belső csúcsban kulcsként is szerepel. Amikor ezt a belső csúcsot elérjük, a keresés mindkét részfában folytatódik tovább. Ennek oka, hogy a 15 a *H* végpontja és egyben az *F* kezdőpontja is. Az 5.8 ábráról leolvasható, hogy az output listába, az *A* címke mellett, helyesen bekerül a *H* és az *F* címke is. Eredményül kapjuk, hogy mindhárom intervallum tartalmazni fogja a 15-ös értéket és csak ezekbe fog beleesni.



5.8 ábra: pont tartalmazás vizsgálata első változat esetén

Második változat esete: Most a szegmensfa második ábrázolási módja esetén mutatom be a pont tartalmazás megoldását.

A továbbiakban tekintsük a felépítésnél említett példát. Kereséskor a gyökérből indulunk, és valamely levélig kell eljutni úgy, hogy a keresett értéket a fa minden szintjén a keresési kulccsal hasonlítjuk össze és annak megfelelően lépünk tovább valamely irányba (balra vagy jobbra) a gyerekcsúcsok felé. Ha pl. azt kérdezzük, hogy melyek azok az intervallumok, amelyek a $q_x = 70$ pontot tartalmazzák, akkor az 5.10 ábrán, a gyökérből indulva jobbra lépünk, majd a 100-as kulcsú csúcsból háromszor is balra lépve jutunk el a megfelelő levélhez. Közben a *C* és *D*, majd a *B* címke az output listába kerül, mivel ezek a keresett intervallumok azonosítói.



5.9 ábra: pont tartalmazás vizsgálata második változat esetén

Mindkét keresési változatban a keresés műveletigénye $O(\log n)$. Ha egy útvonal bejárásakor k db. címke kerül az output listába, akkor ezek kigyűjtésével arányosan nő a műveletigény $O(\log n + k)$-ra.

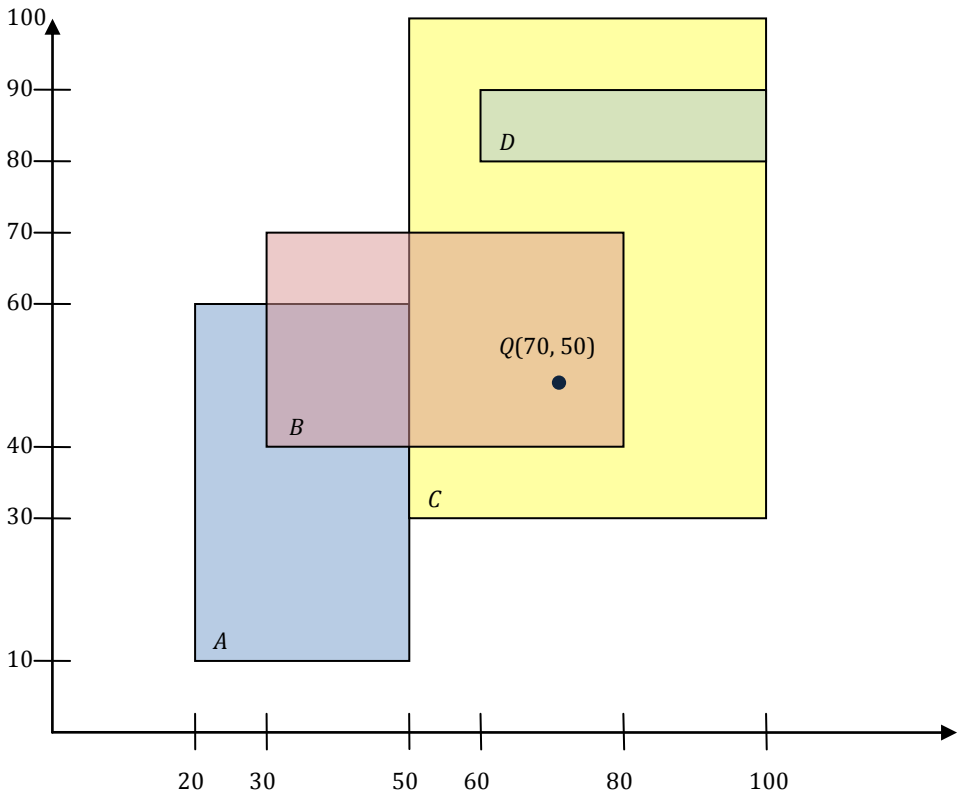
5.5.3 Pont tartalmazás kiterjesztése magasabb dimenzióra (2D)

A továbbiakban csak a második verzióval dolgozom tovább. A feladatot és megoldását az előző példa folytatásaként mutatom be. Legyenek *A*, *B*, *C* és *D* olyan téglalapok, amelyek vízszintes oldalainak *x* tengelyre eső vetületei rendre megegyeznek az előző példában szereplő intervallumokkal (5.10 ábra).

A téglalapok függőleges oldalainak az *y* tengelyre eső vetületei a következő intervallumokat határozzák meg: *A*[10, 60], *B*[40, 70], *C*[30, 100] és *D*[80, 90].

Ha feltesszük azt a kérdést, hogy mely téglalapok tartalmazzák a *Q*(70, 50) pontot, akkor ehhez fel kell építeni egy kétszintű szegmensfa-rendszert. A rendszer alapját az 5.11 ábrán látható szegmensfa képezi. Ennek minden olyan csúcspontjához, amelyben szerepel intervallumazonosító címkeként, külön egy szegmensfát építünk fel az adott csúcsban szereplő címkék által meghatározott intervallumokból. Pl. a 80-as kulcsértéket

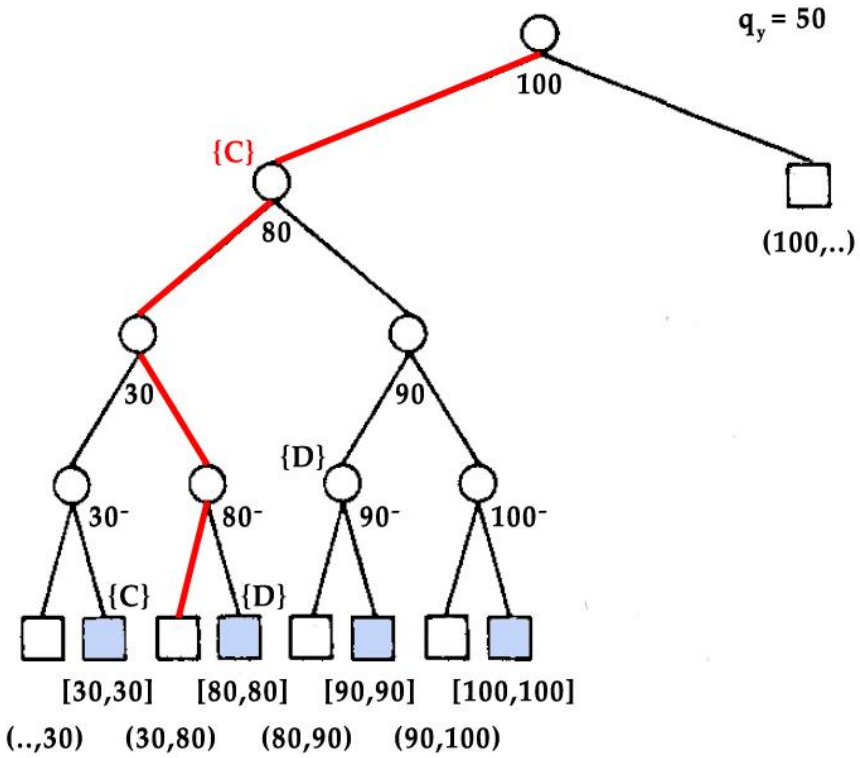
tartalmazó csúcshoz az 5.11 ábrán látható szegmensfa épül fel a *C* és *D* intervallumokból.



5.10 ábra: síkbeli alakzatok 2D szegmensfa építéséhez

Az egy dimenziós keresést úgy módosítjuk, hogy ha címkézett csúcsot érünk el, akkor megállunk és áttérünk a másodlagos fára. A másodlagos fában a pont második koordinátájának megfelelő egy dimenziós tartalmazási keresést hajtjuk végre. Példánkban azt határozzuk meg, hogy melyek a $q_y = 50$ pontot tartalmazó intervallumok. Látható, hogy a keresés útvonala érinti a *C* azonosítót, a *D*-t viszont nem. Ezt úgy kell értelmezni, hogy az elsődleges szegmensfában szereplő *C* és *D* téglalapok közül csak az előbbi tartalmazza a keresett *Q* pontot. Mivel a *B* téglalapra is – itt nem részletezett, de ugyanilyen módon – tartalmazás állapítható meg, így a síkbeli kérdésre a végső válasz az, hogy a keresett *Q* pontot a *B* és a *C* téglalapok tartalmazzák. Az algoritmus eredménye egy output listát szolgáltat.

Meggondolható, hogy a téglalapokra megfogalmazott 2D pont tartalmazás esetén a kétszintű szegmensfa-rendszer helyfoglalása $O(n * \log n)$ lesz, a lekérdezés műveletigénye pedig $O(\log^2 n + k)$, ahol *k* az output téglalapok száma.



5.11 ábra: 2D pont tartalmazás vizsgálata második változat esetén

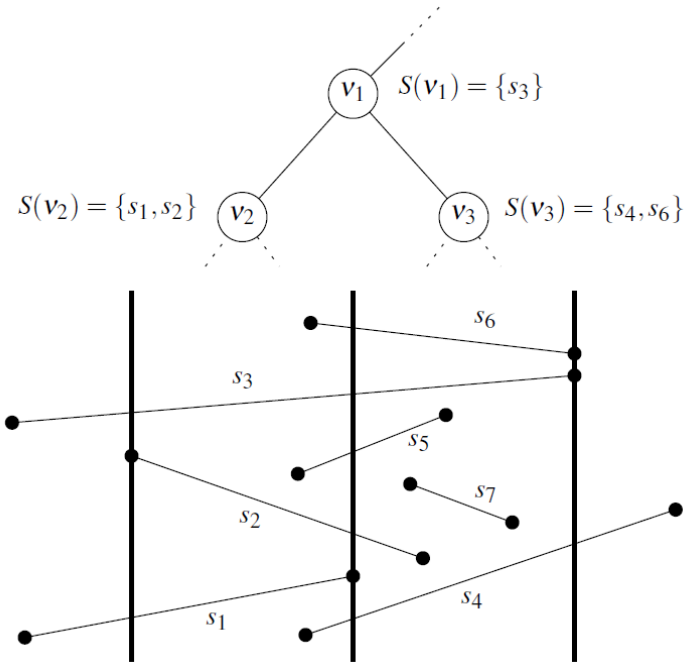
Ha valaki kezdő az adatstruktúrák világában, könnyen arra gondolhat, hogy egyszerűbb lenne két független szegmensfát építeni a téglalapok vízszintes és függőleges oldalaiból, mint intervallumokból, azokban függetlenül keresni, majd a megoldásként kapott listák metszetét venni. Ez a kétségkívül egyszerűbb eljárás hatékonyságban elmaradna a most ismertetett módszertől. A két output listát ugyanis rendezni kellene ($\Theta(k * \log k)$ időben), a közös megoldások $O(k)$ idejű megtalálásához. Feltételezzük, hogy a teljes szegmensfa-rendszert előzetesen felépítjük.

Egy másik lehetséges eljárás lehet, ha kizárólag az x koordináta tartalmazására futtatott keresés eredményeként kapott téglalapok másik oldalából építenénk dinamikusan egy újabb szegmensfát. Ezt követően ebben már csak azokat a vetületeknek tárolnánk, amelyek megfelelnek az első (x koordinátára) kereséskor létrejött output elemeinek. Ha az így létrejött fában hajtanánk végre a keresés algoritmusát, a pont y koordinátájára, ismét nem kapnánk hatékony megoldást. A módszer dinamizmusa miatt, minden keresés alkalmával a második lépésben más-más szegmensfát kellene felépítenünk. Ez a műveletek költségét megnöveli és a hatékonyságot csökkenti.

5.5.4 Szegmensek metszetének vizsgálata

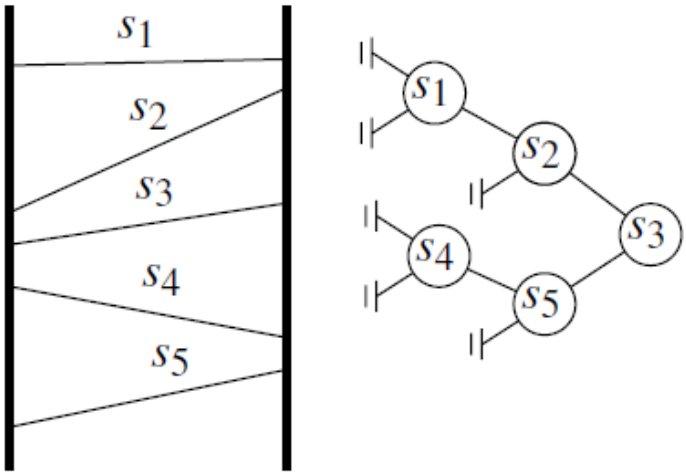
A térinformatikában és a számítógépes grafikában gyakran felmerül a metsző szegmensekre vonatkozó kérdés. Adott a síkon n számú $s_1, s_2, \ldots, s_n$ szakasz, amelyek páronként nem metszik egymást. Ezeket szokás diszjunkt szegmenseknek is nevezni. (Mindössze annyi engedmény tehető egy későbbi meggondolás alapján, hogy a szegmenseknek lehetnek közös végpontjaik is, a szakaszok belsejének (*interiors*) a diszjunkt tulajdonsága azonban nem engedhető el.) Adott ezeken kívül egy függőleges állású q „lekérdező” szegmens. A feladat az, hogy határozzuk meg a szegmensek közül azokat, amelyek metszik a lekérdezés argumentumában szereplő q szegmenst.

A kérdés megválaszolásához újra egy kétszintű adatszerkezetet használunk. Első lépésben az $s_1, s_2, \ldots, s_n$ szegmenseknek az x tengelyre eső vetületeiből, mint intervallumokból szegmensfát építünk. A szegmensfa csúcsai – az felépítésnél ismertetett módon – intervallumokat reprezentálnak: a levelek az elemi intervallumokat, a belső csúcsok pedig a gyerekeiben található intervallumok unióját. A függőleges q szegmens vetülete az x tengelyre egyetlen q_x pontot eredményez. Ezzel, mint „query” ponttal végrehajthatunk egy pont-tartalmazás lekérdezést, a korábban ismertetett módon. Az eredményként kapott intervallumok olyan szegmensekhez tartoznak, amelyeknek lehet metszése a q szegmenssel, de az is lehet, hogy a q alatt vagy felett helyezkednek el a síkon (5.12 ábra).



5.12 ábra: szegmensek metszetének vizsgálata

A metszés eldöntéséhez az adatstruktúra második szintje ad lehetőséget. Ahogyan a 2D pontok tartalmazásának vizsgálata esetén, itt is újabb fastruktúrát hozunk létre a szegmensfa minden olyan csúcsához, amely valamelyik szegmens azonosítóját tartalmazza. Most azonban nem egy újabb szegmensfát, hanem egy bináris keresőfát rendelünk az ilyen csúcsokhoz. A szegmensfa egy csúcsánál szereplő diszjunkt szegmensek vízszintes vetületei nagyobb vagy egyenlő módon lefedik a csúcs által reprezentált intervallumot, ezért sorba rendezhetők az adott intervallum felett. Ez a rendezettség teremt lehetőséget a keresőfa használatára. Ha végrehajtunk két keresést a q szegmens alsó és felső végpontjának az y koordinátájával, akkor megkapjuk a metsző szegmenseket (5.13 ábra).



5.13 ábra: bináris keresőfa építése szegmensfákhoz

A teljes adatstruktúra helyfoglalása $O(n*\log n)$, a fa felépítése $O(n*\log n)$ időben történhet, a válaszadás ideje pedig $O(n*\log^2 n)$.

5.5.5 Téglalapok metszetének vizsgálata

A következőekben egy olyan feladatot vizsgálok meg, amely téglalapok metszésének a megállapítását kívánja. Adott téglalaprendszer esetén meg kell határozni az összes olyan téglalap párt, amelyek közös résszel rendelkeznek (átfedés, metszet, tartalmazás). A háromból most a metszés két típusát célszerű megkülönböztetni: az átfedés olyan metszést valósít meg, amelyben mindkét téglalapnak van a közös részen kívüli saját területe, míg a tartalmazás esetén az egyik téglalap teljes egészében a másik belsejében található. Az 5.10-es ábrán az (*A*, *B*) és a (*B*, *C*) téglalappárok átfedést valósítanak meg.

(Felfogás kérdése az, hogy az egy él mentén illeszkedő *A* és *C* téglalap párt átfedőnek tekintjük-e vagy sem. Az egyszerűbb tárgyalás érdekében a külső illeszkedést most ne tekintsük átfedésnek.) A (*C*, *D*) téglalappár pedig a tartalmazás esetét mutatja.

Az átfedő téglalapok megállapítása visszavezethető az előző pontban ismertetett eljárásra, amelyet szegmensek metszésére alkalmaztam. Tekintsük ebben a feladatban szegmenseknek a téglalaprendszer minden vízszintes oldalát, lekérdező szegmensnek pedig rendre a téglalapok függőleges oldalait. Ha ezekkel a függőleges oldalakkal sorban, egymás után lekérdezést hajtunk végre a vízszintes oldalak szegmensfájában, az előző alfejezetben megadott módon, akkor megkapjuk azokat a párokat, amelyek metszhetik egymást. Mivel egy téglalap oldal adott méretű, ezért az így kapott eredményeket még szűrni kell, mert a lekérdezés eredménye egy egyenesre vonatkozik.

A végleges metszés tényét pedig meg lehet állapítani a vízszintes oldalak magasságaiból épített kiegyensúlyozott bináris keresőfa segítségével. Nem kell mást tenni, mint – minden egyes függőleges oldal esetén – a lekérdező oldal két végpontjával végrehajtani egy-egy keresést a bináris keresőfában. A két végpont közé eső vízszintes oldalak, azok, amelyekhez a metsző téglalapok tartoznak.

Ha egy vízszintes és egy függőleges oldalpár metszi egymást, akkor a hozzájuk tartozó téglalapok nyilvánvalóan átfedők. Egy átfedő téglalap párnak kettő, vagy négy metsző oldalpárja van, így egy átfedést többször is megkapunk. A válaszadás ideje még így is $O(\log^2 n + k)$, ahol *k* az átfedő téglalap párok száma.

A teljes tartalmazások felderítése, a metszetek vizsgálatához képest, valamivel már összetettebb eljárást igényel. A lekérdező függőleges oldalakat most nem használhatjuk egymástól függetlenül, hanem egymás után, párosával vesszük őket. Azaz, egy vízszintes téglalapoldal két végpontjával sorban végrehajtunk két keresést a szegmensfában. A két lekérdezés eredményéből csak azokat a vízszintes oldalakat tartjuk meg, amelyekbe a lekérdező oldal mindkét végpontja beleesik. Ha ezen, vízszintes oldalakról áttérünk a hozzájuk tartozó függőleges oldalakra, akkor hasonló értelmű teljes tartalmazást kérdezünk le a bináris keresőfákban. Egy kevés járulékos konstans hely- és időigényű adminisztrációval az összes tartalmazó téglalap párt meg tudjuk határozni. A válaszadás ideje éppen úgy $O(\log^2 n + k)$, mint az átfedés esetén.

A műveletigény a gyakorlat szempontjából jelentős felső becslést tartalmaz, ugyanis a szegmensfa csúcsaihoz rendelt bináris keresőfák pontszáma általában alacsony; a legritkább esetben *n* nagyságrendű.

Megjegyzem, ha csupán két téglalap kölcsönös helyzetét kellene megállapítani, akkor eljárhatnánk úgy, hogy mindkét téglalap, minden csúcsára megvizsgálnánk azt a kérdést, hogy a másik téglalap belsejében helyezkedik-e el. Ha minden válasz nemleges, akkor a két téglalap diszjunkt. Ha az egyik téglalap minden csúcsára pozitív választ kapunk, akkor az a másik téglalap belsejében fekszik. Végül, „vegyes” válasz esetén a két téglalap átfedő.

5.6 Konvex alakzatok indexelése

A térinformatikában és a számítógépes grafikában fontos szerepet játszanak a konvex alakzatok. A térinformatikát tartva szem előtt, továbbra is a síkbeli objektumok világában maradunk. Egy görbe vonalakkal határolt alakzat is lehet konvex, itt azonban egyenes szakaszokkal határolt konvex poligonra gondolunk. Egy térképi objektum vektorizálása útján egyenes szakaszokból álló poligonhoz jutunk, amelyet szükség esetén konvex részekre bonthatunk. Egy másik lehetőség a konvex alakzatok megjelenítésére az, ha egy objektum befoglaló téglalapja helyett valamilyen befoglaló konvex burkot tekintünk. Ennek elkészítése nagyobb befektetést kíván, de a pont-tartalmazási és a metszési kérdések eldöntésénél jóval megbízhatóbb válaszokat kapunk. Ha pl. adott *A* és a *B* térképi alakzatok valójában nem metszik egymást, azért a befoglaló téglalapjaik még könnyen mutathatnak metszést (különösen akkor, ha a szóban forgó alakzatok szabálytalan, amőba-alakú határvonallal rendelkeznek). Ha azonban egy-egy viszonylag jól illeszkedő befoglaló konvex poligonra térünk át, akkor a hamis találatok esete viszonylag ritka lesz.

Az 5. fejezetben bemutatott eddigi struktúrák az objektumok befoglaló téglalapjaira épülnek. Meggondolható, hogy a konvex burokba foglalás elméletét követve, lehetne olyan R-fákat is építeni, amely a legkisebb befoglaló téglalap helyett a legkisebb konvex burkokkal dolgozna, és azok kerülnének tárolásra mind a levelekben, mind a belső csúcsokban. Itt jelentkezik az első példa arra, hogy kereséskor nehéz poligon esetén számolni tartalmazást, de ha mód nyílik rá, akkor sokkal pontosabb megoldásokhoz juthatunk.

A továbbiakban konvex poligonok rendszerét tekintem, és azzal már nem foglalkozom, hogy ezek esetleg milyen eredeti alakzatok befoglalói. Az így kapott konvex alakzatokat is indexelni kell valahogyan, pl. befoglaló téglalapjaik alapján, de ebben az alfejezetben nem ezen lesz a hangsúly, mert a pontosabb eredmények

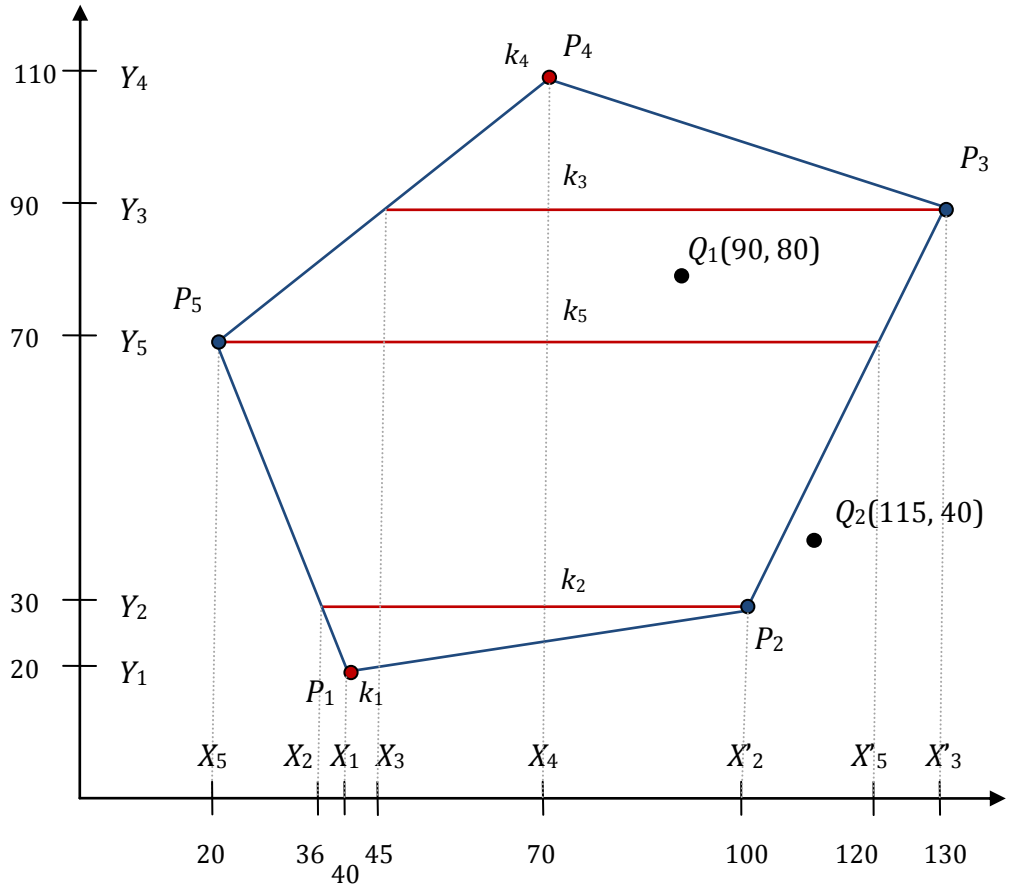
eléréséhez érdemes egyéb, további lehetőségeket is megvizsgálni. A konvex burok önmagában egy nehezen kezelhető eszköz az indexelésre, mert azok tárolásakor sok koordinátával kellene dolgozni, így a vizsgálat most az egyes alakzatok egy lehetséges felbontására, és a részek indexelésére irányul. Egy kézenfekvő felosztás lehet pl. a szakirodalomból jól ismert Delaunay-háromszögekre való bontás.

Dolgozatomban, a vizsgálat középpontjában egy másik lehetőség áll, nevezetesen az, amikor a konvex alakzatot az *x* tengellyel párhuzamos (vízszintes) vonalakkal trapézokra bontjuk. Ezek a trapézok lesznek azok az elemi egységek, amelyeket egy index-struktúrában el kell helyezni. Azt vártam, hogy a trapézoldalakból épített szegmensfa, valamint a magasságaikból épített további adatszerkezetek, pl. bináris keresőfák segítségével hatékony megoldást lehet találni a ponttartalmazási és az alakzatmetszési alapfeladatokra. Célkitűzésem megvalósítása azonban részben negatív eredményt hozott. Erre a ponttartalmazás feladatának megoldásánál fogok mélyebben rávilágítani. Most előzetesen csak annyit jegyeznék meg, hogy a trapézokra történő felbontáshoz nem illeszkedik a szegmensfa alkalmazások mögött álló ún. *plane sweeping* (a sík söprésének) módszere, amely egy függőleges egyenessel halad végig (balról jobbra) az alakzatokat tartalmazó síkon, vagy ablakon. Egy trapézhoz olyan geometriai vizsgálat a megfelelő, amely a ferde oldalak metszéspontján átfektetett egyenessel „söpri” a trapéz-idomot. A trapézokra bontás ötletét azonban nem vetettem el véglegesen, hanem megpróbáltam valamilyen pozitív állítást megfogalmazni a használhatóságára. Az eredmények, amelyek ilyen értelemben „árnyalják” a trapézalapú indexelés lehetőségét, arra mutatnak, hogy a befoglaló téglalapok alkalmazása még egy alakzat felbontása esetén is megfelelőbb eszköz lehet.

A trapézokra bontásra vonatkozó elvi döntés még számos megvalósítási lehetőséget hagy nyitva. Ezeket mind sorra veszem, előbb azonban egy példával szeretném bemutatni az alap jelenséget. Az 5.14. árán egy *K*-nak nevezett konvex ötszög látható, amely pontjainak felsorolásával adható meg: $P_1(40, 20)$, $P_2(100, 30)$, $P_3(130, 90)$, $P_4(70, 110)$, $P_5(20, 70)$. Ez a sokszög olyan, hogy bármely pontpárjára igaz az, hogy nincsenek egy vízszintes, vagy egy függőleges egyenesen, vagyis a legáltalánosabb eset bemutatására éppen alkalmasak.

Ha felszeleteljük a *K* sokszöget a csúcsain áthaladó, vízszintes egyenesekkel, akkor az ötszög tartományán belül minden csúcson át egy trapéz oldala halad, amely a P_1 és a P_4 pontok esetében pontszerű lesz (háromszögeket kapunk). Tekintsük most ezeket is intervallumnak, és az alsó és felső háromszögeket is trapéznak. Ekkor át lehet térni egy

új ábrázolási módra, amely a trapézoldalak, mint intervallumok, valamint azok magasságainak megadásával valósul meg. Ezek rendre: $k_1[40, 40]$, $k_2[36, 100]$, $k_3[45, 130]$, $k_4[70, 70]$, $k_5[20, 120]$ és $Y_1 = 20$, $Y_2 = 30$, $Y_3 = 90$, $Y_4 = 110$, $Y_5 = 70$ (5.14 ábra).

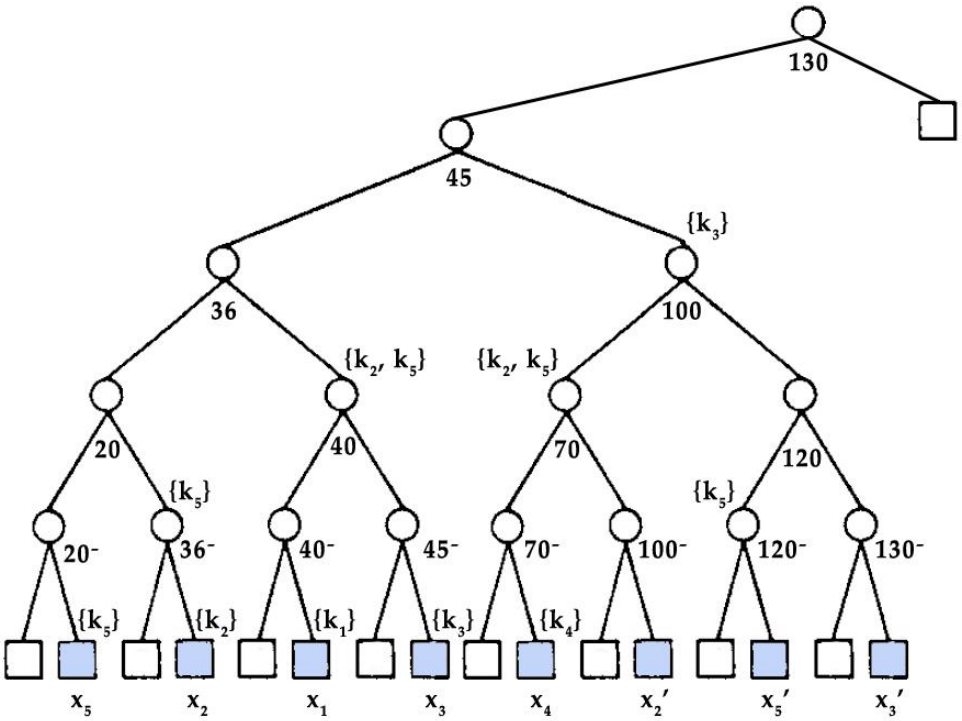


5.14 ábra: poligon trapézokra bontása

A k_1 – k_5 intervallumok az x tengelyen nyolc osztópontot határoznak meg. Ezek sorban: 20, 36, 40, 45, 70, 100, 120 és 130. A nyolc osztópont 17 elemi intervallumot hoz létre, ahogyan ezt korábban a szegmensfák esetén is láttuk. Az elemi intervallumok előállítása után már felépíthető a szegmensfa (5.15 ábra).

A szegmensfa csúcsai (a korábban ismertetett módon) intervallumokat reprezentálnak, illetve keresési kulcsokat tartalmaznak. A k_1, k_2, …, k_5 trapézoldalakat, mint címkéket, elhelyezzük a szegmensfa csúcsainál, ezzel segítve majd a kereséseket.

A k_1 – k_5 trapézoldalak magasságait egy másodlagos adatszerkezetben, egy kiegyensúlyozott bináris keresőfában tároljuk (*BBST – Balanced Binary Search Tree*). A fa struktúrától el lehet térni, de optimálisnak ez tekinthető.

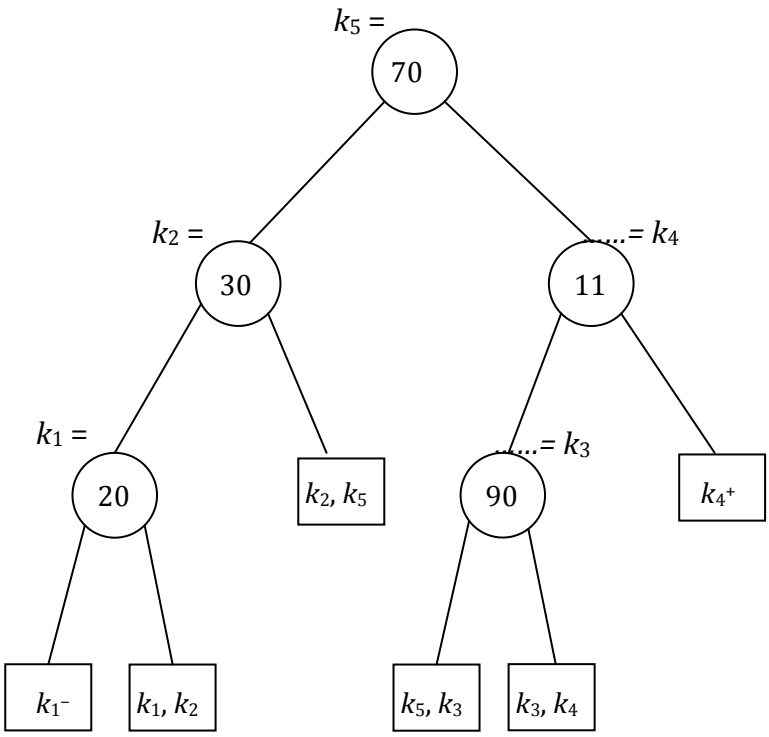


5.15 ábra: poligon trapézokra bontásából szegmensfa létrehozása

Két megoldás is kínálkozik a keresőfa alkalmazására, de erre a kérdésre még később visszatérek.

Lehetne egy olyan megoldást választani, hogy a szegmensfa címkézett csúcsaihoz külön-külön létrehozunk egy-egy keresőfát az ott szereplő trapézoldalak magasságaira. Azonban, ha nem túl sűrű a felosztás és az egyes csúcsokban nem sok magasság címke szerepel, akkor érdemes az egész K konvex alakzatra egyetlen keresőfát konstruálni, amelyben minden trapézoldal magassága megtalálható. Az 5.16. ábrán látható keresőfa annyiban speciális, hogy nem csak magukat a 20, 30, 70, 90 és 110 magasságértékeket lehet bennük keresni, hanem bármely keresési értékre megadja, hogy – amennyiben az nem egyenlő valamelyik magasságértékkel, akkor – hol helyezkedik el a magasságértékek rendezett sorában.

Pl. továbbra is az 5.16 ábránál maradva, a $k = 40$ egyik magasságértékkel sem egyezik meg, így a fában való kereséskor biztosan le kell menni egy levélig (jelen esetben a 30-as csúcs alatti levélcsúcsig). Így kapjuk eredményül, hogy a $k = 40$ a k_2 és a k_5 közötti sávban helyezkedik el. A $k = 120$ keresésekor a 110 csúcs alatti levélig jutunk el, amelyben szereplő k_4^+ jelentése, hogy a 120, mint egy pont y koordinátájának értéke, a k_4 trapézoldal felett található.



5.16 ábra: BBST fa a trapéz magasságokhoz

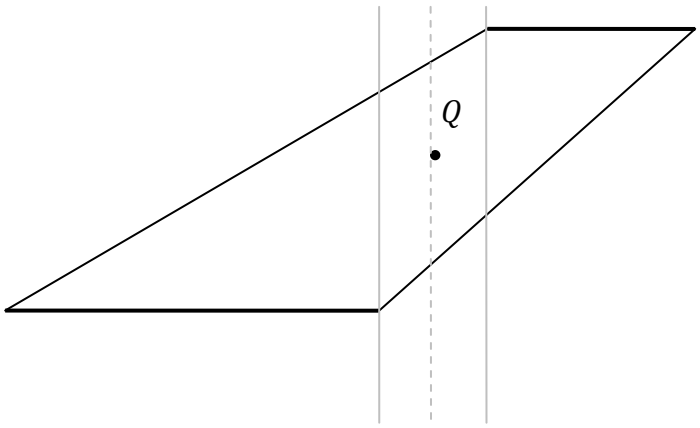
Ezzel eljutottam oda, hogy ennek az indexelési felfogásnak a különböző változatait sorra vegyem. Előbb azonban nézzük meg azt, hogy milyen kérdésekre adhatunk választ ebben az indexelési struktúrában.

5.6.1 Pont tartalmazás vizsgálata

Első lépésben azt a kérdést célszerű megválaszolni, hogy a K konvex alakzat a belsejében tartalmaz-e egy adott $Q(x, y)$ pontot. Ez a kérdés már korábban, a szegmensfáknál felvetett téglalapok esetében is szerepelt valamivel általánosabb formában, mivel ott minden tartalmazó téglalapra kíváncsiak voltunk. Most azért szűkítendő a kérdés egy alakzatra, mert az maga is részekre van felbontva. (Ha alakzatok egy teljes halmazára szeretnénk feltenni a ponttartalmazás kérdését, akkor előbb egy külső ciklusban meg kell vizsgálni pl. az alakzatok befoglaló téglalapjaira a kérdést, a már megismert módon. Ha bizonyos téglalapoknál negatív választ kapunk, akkor finomításképpen fordulhatunk a befoglaló konvex alakzatokhoz.)

Először a vízszintes trapézoldalak szegmensfájából indultam ki, ami jónak is tűnt egészen addig, amíg az 5.17 ábrán látható példa elő nem került. Ez azt mutatja, hogy egy pont úgyis lehet egy trapéz belsejében, hogy az x koordinátája a két oldal diszjunkt vetülete közé esik. Ez a „bonyodalom” áthidalható ugyan, de célszerűnek látszott előbb

a másik nézetből megközelíteni a problémát, azaz elsőbbséget kap a vizsgálatban a trapézoldalak magasságaiból alkotott bináris keresőfa, amelyet eredetileg másodlagos indexstruktúraként definiáltam.



5.17 ábra: Trapéz az oldalakat nem metsző „söprő” egyenessel

Hajtsunk végre egy keresést az alakzathoz tartozó bináris keresőfában a *Q* lekérdező pont *y* koordinátájával. Ennek eredményeképpen megkapjuk azt, hogy hol van az y helye a trapéz oldalmagasságok rendezett sorában, pontosabban, hogy melyik két magasságérték közé esik. A geometria nyelvén értelmezve a keresés eredményét, azt tudjuk meg, hogy a *Q* pont *K* alakzat alatt, vagy afelett van, vagy pedig az alakzat magasságában helyezkedik el, és ebben az utóbbi esetben melyik az a trapéz, amelynek a sávjába esik. Ezzel a kereséssel egy finomítást végeztünk és megkapjuk, hogy érdemes-e vízszintes irányban is tovább vizsgálni a tartalmazás lehetőségét.

Tegyük fel, hogy a $Q(x, y)$ pont a *K* konvex alakzat felbontásakor létrejött *T* trapéz belsejébe esik, amelynek k_1 és k_2 a két vízszintes oldala. Az 5.18 ábrán látható jelöléseket használva, felírhatjuk a (valódi) ponttartalmazás alábbi feltételét:

$$Q \in T \Leftrightarrow \bar{x} < x < \bar{x}'$$

Fejezzük ki $\bar{x}$ -t és $\bar{x}'$ -t:

$$\bar{x} = \frac{x_2 - x_1}{y_2 - y_1} * (y - y_1) + x_1$$

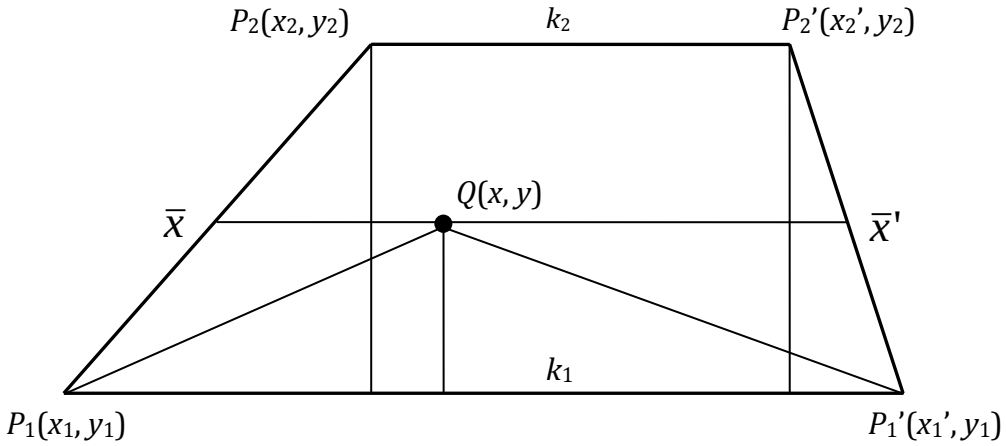
$$\bar{x}' = \frac{x_2' - x_1'}{y_2 - y_1} * (y - y_1) + x_1'$$

Átrendezve kapjuk a ponttartalmazás következő két feltételét.

$$\frac{x_2 - x_1}{y_2 - y_1} < \frac{x - x_1}{y - y_1}$$

$$\frac{x_2 - x_1'}{y_2 - y_1} < \frac{x' - x_1'}{y - y_1}$$

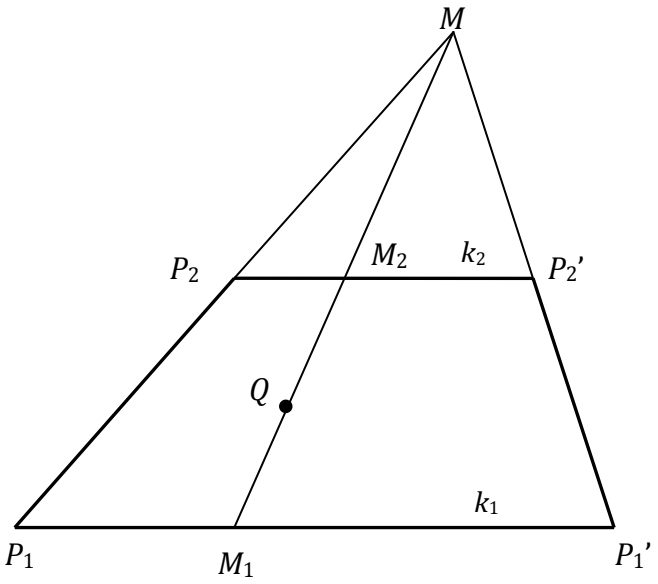
A két képlet geometriai értelmezése nyilvánvaló: azt fejezik ki – derékszögű háromszögek oldalainak az arányával, vagy a tangens szögfüggvény segítségével –, hogy Q pont a $P_2P_1P_2$' szög és a P_1P_1'P_2' szög tartományán belül helyezkedik el.



5.18 ábra: Trapéz belső ponttal

A bemutatott eljárás műveletigénye $O(\log n)$, mivel a kiegyensúlyozott bináris keresőfát $2n + 1$ csúcsból építettük fel. Hangsúlyozom, hogy a műveletigény egy alakzat esetére vonatkozik, szemben a korábbi, hasonló kérdéssel, amelyet a teljes téglalap-rendszerre tettem fel. Megjegyzem még, hogy a bináris keresőfa előzetes felépítése $\Theta(n * \log n)$ lépésszámot igényel, azt azonban számos lekérdezés követheti. Az eljárás alkalmazásával az 5.14 ábrán szereplő K konvex alakzatra, valamint a Q_1 és Q_2 pontokra azt kapjuk, hogy $Q_1 \in K$ és $Q_2 \notin K$.

Eddigi megállapításaim alapján már rávilágíthatok a trapézokhoz illeszkedő síksöprés módjára is. Tekintsük ehhez az 5.19 ábrán a P_1P_1'M háromszöggé kiegészített 5.18-as ábrán látható trapézt. Világos, hogy a Q ponton át nem a függőleges, hanem az M ponton átmenő söprő egyenes lesz az, amely mindig metszi a k_1 és a k_2 trapézoldalakat, ha Q a trapéz belsejében fekszik.



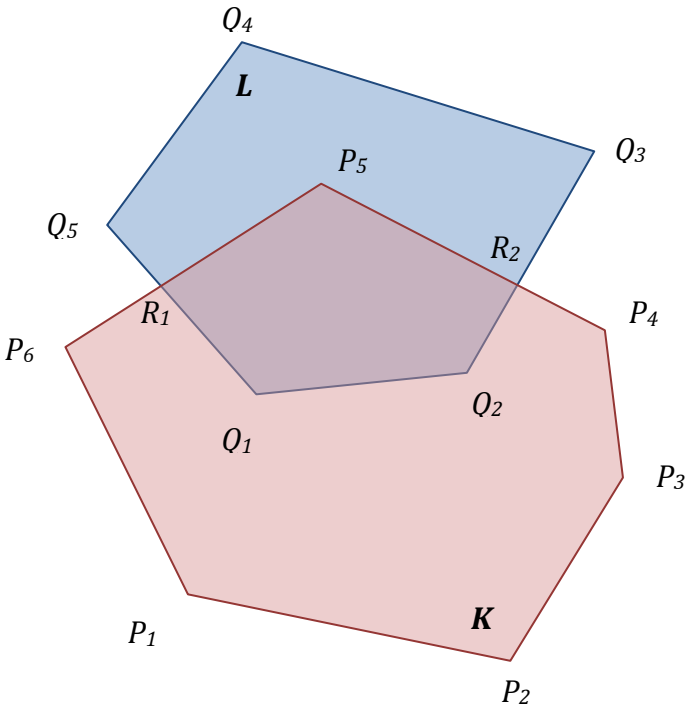
5.19 ábra: A sík söprésének elvi háttere trapéz esetén

Egy ilyen síksöprésre nem építethetünk szegmensfát. Először is, minden egyes trapézszeletnél külön munkát jelentene a ferde trapézoldalak *M* metszéspontjának meghatározás, a *QM* egyenes egyenletének felírása, és a két vízszintes trapézoldallal való metszés megállapítása. Továbbá, módszerünk nem lenne elég általános, mert csak ismert *y* koordinátával rendelkező pontok esetén lenne alkalmazható. Ha tekintenénk egy olyan egyenest, amelyet csak az *x* koordinátája határoz meg, akkor nem tudnánk értelmezni a szegmensfa építésének eljárását.

5.6.2 Alakzatok metszetének vizsgálata

A metszés (átfedés és tartalmazás) kérdését téglalapok egy halmazára vizsgálják. A megoldást visszavezetik a szegmensek metszésére, ahol a téglalap oldalak valósítják meg az egyes intervallumokat. Az eljárás ismertetésénél megjegyeztem, hogy amennyiben csak két téglalap metszését kívánjuk megállapítani, akkor a feladat visszavezethető a ponttartalmazás vizsgálatára, ahol a két téglalap pontjaira lekérdezzük azt, hogy a másik téglalap tartalmazza-e azokat. Tovább idézve a megjegyzést, a kapott válasz három kategóriába sorolható: a két téglalap lehet diszjunkt, átfedő és tartalmazó viszonyban egymással.

Áttérve a konvex alakzatok esetére, ismét azt lehet mondani, hogy amennyiben az alakzatok egy halmazára szeretnénk a páronkénti metszést megállapítani, akkor előbb vizsgálódhatunk a befoglaló téglalapok halmazán, és a válasz finomítása céljából térhetünk át a (befoglaló) konvex alakzatokra.



5.20 ábra: Metsző konvex alakzatok

Tekintsünk két konvex alakzatot, például az 5.20 ábrán látható *K*-t és *L*-et, amelyek metszési viszonyára vagyunk kíváncsiak. Ezúttal is a ponttartalmazás vizsgálatához nyúlhatunk. Hajtsuk végre a *K* alakzat $P_1 - P_6$ pontjaival a tartalmazás lekérdezését az *L* alakzatra nézve, majd pedig az *L* alakzat $Q_1 - Q_5$ pontjaival tegyük ugyanezt a *K*-ra nézve. Ha csupán a metszés kérdése érdekel bennünket, akkor elegendő az első pozitív válaszig folytatni a vizsgálatot, ami esetünkben a P_5 pontnál áll elő.

Ha a teljes vizsgálatot végrehajtjuk, akkor példánkban a $\{P_5, Q_1, Q_2\}$ ponthalmazt kapjuk. Ha ezt kiegészítjük az oldalpárok metszésének vizsgálatával, akkor az R_1 és R_2 pontokat is megkapjuk. Így előállt a konvex közös rész csúcspontjainak $\{Q_1, Q_2, R_2, P_5, R_1\}$ halmaza. A közös rész meghatározásának lehetőségét csak érzékeltetni kívántam, a probléma részletesebb ismertetésétől eltekintek.

A teljes vizsgálat műveletigényének meghatározásához tegyük fel, hogy a *K* alakzat *m* csúcsot tartalmaz, az *L*-et pedig *n* számú pont feszíti ki. Ekkor a két kiegyensúlyozott bináris keresőfában $O(m * \log n + n * \log m)$ lépésben keresünk.

5.6.3 Trapézfelbontás további lehetőségei

Az 5.21. ábrán bemutatott felbontás az eredeti konvex sokszöget egzakt módon szeleteli trapézokra. A vízszintes trapézoldalakat, a poligont szeletelő egyeneseket mindig a

sokszög csúcsain át húztam be. Ennek a következménye, hogy a trapézok magassága változó, ezért valamelyest bonyolulttá teszi a lekérdezéseket, hiszen keresőfát és számítási képleteket kellett alkalmazni.

Egyszerűbb megoldáshoz juthatunk, ha ekvidisztans (egyenlő távolságú) szeletelést alkalmazunk a konvex poligonon. Ekkor azonban egy vízszintes sávban nem feltétlenül kapunk trapézt, hiszen előfordulhat, hogy az eredeti sokszögnek valamelyik csúcsa éppen egy ilyen sávba esik, akkor ott nem egy „ferde” trapézoldal vezet a sáv alsó és felső oldala között, hanem egy csúccsal „megtört” szög alakul ki. Ilyen törést okoz az 5.21 ábrán az alsó 20 – 40-es grid-sávban a P_2 pont, vagy feljebb a P_5 és P_3 pontok is. Ebben az esetben több lehetőségünk is van arra, hogy milyen közelítést alkalmazunk az egyszerűbbé tételre. Az alkalmazástól függően választhatunk pl. az alábbi lehetőségek közül, amelyeket nem dolgozok ki részletesen, hanem csak, mint további elméleti lehetőségeket vetek fel:

(1) Ha a trapézrendszert közelítésre használjuk egy elég finom felosztás esetén, vagyis nem ragaszkodunk annak befoglaló jellegéhez, akkor a kialakult kis „kicsúcsosodó” háromszögeket le lehet vágni. Így olyan trapézrendszert kapunk, amely továbbra is konvex lesz, hiszen konvex alakzatból alakítottuk ki lemetszésekkel, de a terület csak közelítő lesz, mert a lemetszésekkel minimális területet, de mégis levágtunk valamekkorát. Az 5.21 ábrán egy ilyen lemetszéssel keletkezett trapézt láthatunk (piros színnel jelölt). Ez a közelítés bizonyos esetekben nem ad megoldást olyan lekérdezésre, amely egy pont tartalmazására vonatkozik és a pont éppen a konvex poligon levágott részébe esik. Hiszen a levágás miatt a lenyesett részekben nem fogjuk a pontot keresni. Meggondolandó, hogy erre milyen hatékony megoldást lehetne adni.

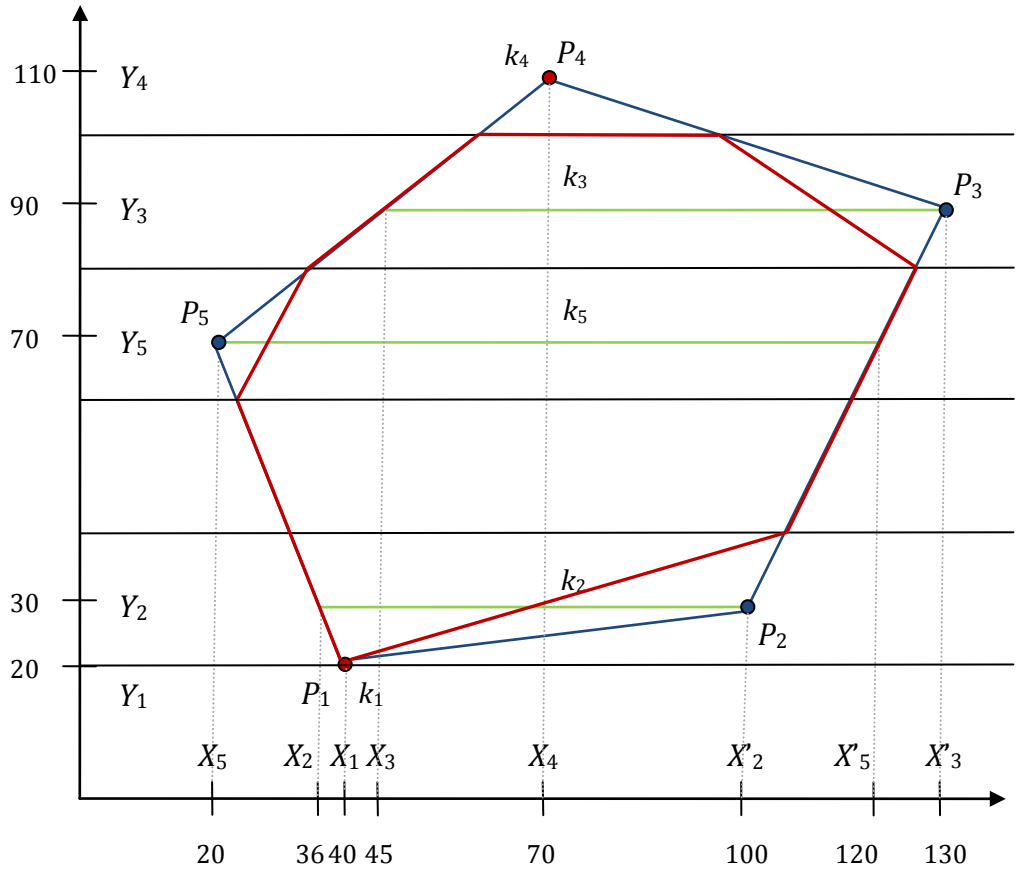
(2) Ha az alakzat befoglaló jellegéhez ragaszkodunk, akkor nem lehet a lemetszés műveletét alkalmazni rá az (1) pontban leírt tartalmazási probléma miatt. Helyette csak bővíteni lehet azt. Nem nyilvánvaló az, hogy hogyan lehetne a szóban forgó esetekben az egyes szeleteken belül úgy kialakítani a trapézokat, hogy az egész rendszer-illeszkedő maradjon. Ezt a lehetőséget nyitva hagynám, vagy inkább nagy valószínűséggel elvetném.

(3) Ha a trapézok helyett azok befoglaló téglalapjaikra térnénk át az egyenlő magasságú sávokban, akkor visszajutunk a befoglaló téglalapok gondolatához,

finomabb felosztás mellett. A téglalapokat egy szeleten belül, a függőleges oldal, különböző behúzásával az alábbi módokon lehet kialakítani:

- befoglalóan,
- beírt módon, és
- átlagoló jelleggel.

Az egyenlő magasságú szeletek alkalmazása, mint látjuk, az eredeti alakzat egy közelítéséhez vezet, viszont a függőleges irányú tájékozódáshoz nem kell kiegyensúlyozott bináris keresőfát építeni a magasságokra, hanem elegendő pl. a legegyszerűbb grid-indexet alkalmazni. De ha egyéb, a munkám során megismert indexstruktúrát választunk, az is éppen olyan megfelelő. Így juthatunk el a vegyes indexelés fogalmáig és lehetőségeihez.

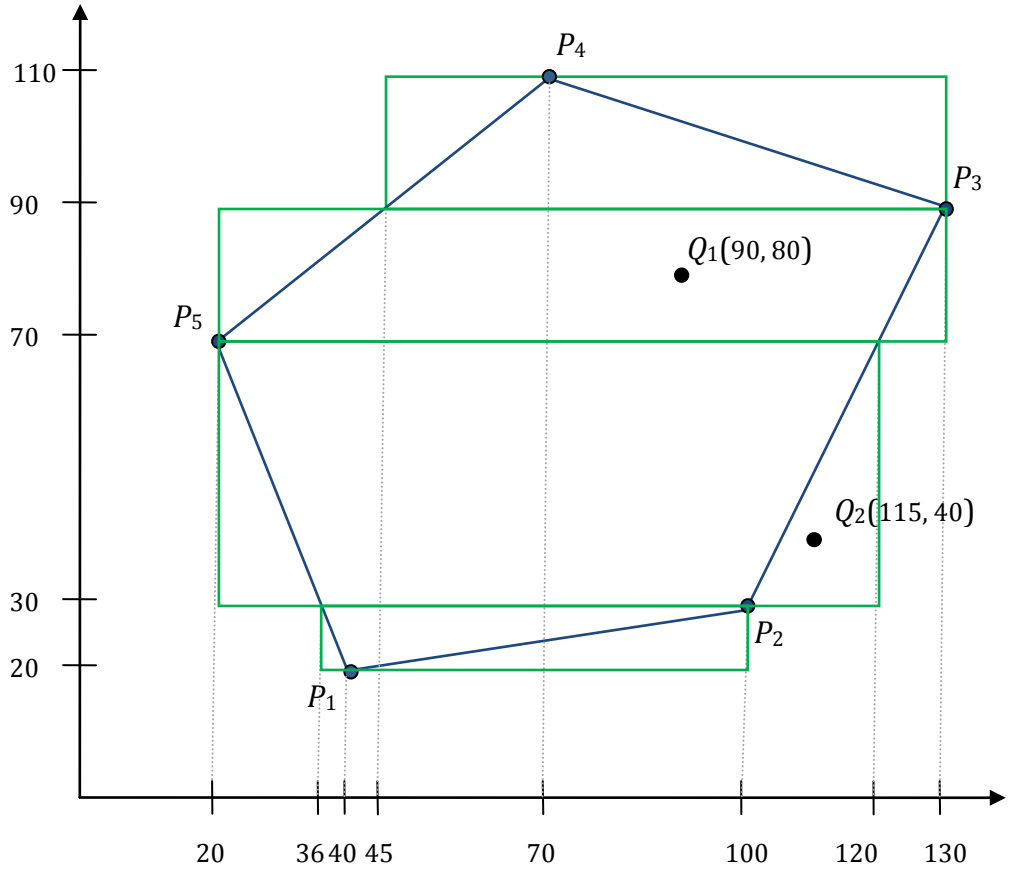


5.21 ábra: Egyenlő magasságú trapézokra bontás

Az alfejezet bevezetőjében szóltam arról a felismerésről, amelyre a pont-tartalmazási feladat vizsgálata során jutottam. A trapézokra szeletelt konvex

alakzatokhoz nem igazán illeszkedik a függőleges egyenesekkel megvalósított síksöprés (*plane sweeping*) eljárása. Az 5.17 ábrán látható jelenség ugyan ritka, de nyílván nem hagyható figyelmen kívül. Ha egy függőleges egyenes áthaladhat egy trapézon úgy, hogy vízszintes oldalait nem metszi, akkor erre a lehetőségre külön vizsgálatokkal minden esetben fel kellene készülni. A természetesebb megoldás az lenne, hogy a trapézokhoz illeszkedő síksöprésre térnénk át, az azonban – ahogyan vázoltam – algoritmikusan bonyolult és költségigényes lenne.

A felmerült problémára egy természetes válasz lenne az, hogy készítsük el a trapéz szeletek befoglaló téglalapjait, és azokkal dolgozzunk! Egy ilyet esetet mutat az 5.22 ábra.



5.22 ábra: Trapéz szeletek minimális befoglaló téglalapjai

Amennyiben a konvex sokszög maga is egy alakzat befoglalója, akkor egy annál kevésbé pontos befoglalóra, a körülírt téglalap halmazra térünk át. Ez azonban jóval szorosabban veszi körül az eredeti alakzatot, mint egyetlen befoglaló téglalap, azaz jobban közelíti azt, és jóval könnyebb vele dolgozni, mint a trapézsávokkal.

5.6.4 Az indexelés időbeli változatai

Először is azt kell megjegyezni, hogy az alkalmazásokban általában nem egyetlen konvex alakzat szerepel, hanem alakzatok egész rendszere. Egy digitális térképen, egy ablakban is, egyszerre számos objektum látszik. Amikor egy kérdést felteszünk, akkor általában nem egy kitüntetett objektumot kell megvizsgálni, hanem mindet a térképen, vagy mindegyiket, amelyek éppen láthatók. Ez arra mutat, hogy az előbbiekben tárgyalt elsődleges és másodlagos index-struktúra fölött van még egy – valójában elsődleges – struktúra, amely az egyes objektumok, mint osztatlan entitások hatékony elérését támogatja.

A térinformatikai alkalmazásoknál mindig eldöntendő az, hogy ha egy indexelést nem az elsődleges tároláshoz alkalmazunk, hanem egy további, finomabb indexelésről van szó, amely feladatosztály megoldását támogatja, akkor azt az előfeldolgozás során létrehozzuk, vagy mindig valós időben, dinamikusan építjük fel. Ebben a dolgozatban általában az előre felépített, „a priori” indexelést vettem alapul.

Meg kell azonban jegyezni, hogy a dinamikus indexelés is járható út. Különösen akkor, hogy ha egyenlő távolságú szeletelést alkalmazunk egy konvex alakzat esetén. Ekkor ugyanis pl. egy pont tartalmazásakor először pl. a létrehozott grid-index alapján vertikálisan tájékozódhatunk, és utána már csak az adott magasságban szereplő objektumokra kell a szegmensfát felépíteni.

6. Mozgó objektumok indexelése

A mai világban a technológiák nagyot léptek előre a mobil kommunikációk és a GPS-ek (*Global Positioning Systems*) területén. Ez a fejlődés a felhasználók figyelmét a 2D térben mozgó objektumokra vonatkozó információk hatékony kezelése felé irányították. Az objektumok a saját aktuális pozícióikat küldik el, amelyek időközönként bizonyos okokból a felhasználóknak is továbbíthatók. Ezeket az információkat tér- és időbelieknek nevezzük, mert az objektumok térbeli elhelyezkedése időben változik.

A korábban bemutatott indexelési eljárások nem alkalmasak mozgó objektumok tárolása. Inkább tekinthetők úgy, mintha csak egyetlen időpillanatban tároltuk volna el a mozgó objektumok éppen aktuális helyzetét. A mozgást is reprezentálva, az adatbázison nagy mennyiségű frissítést kellene végrehajtani, amelyek a műveletigényeket megnövelnék és a hatékonyságot nagyon lecsökkentenék. Ha viszont a mozgó alakzatokat úgy indexelnénk, hogy minden időpillanatbeli állapotot eltárolnánk, akkor drasztikusan megnövekedne az adatbázis mérete. Emellett nincs lehetőség lekérdezni egy tetszőleges időponthoz tartozó helyzetet, kivéve, ha a megadott időpont pont egy mintavételezési pillanat. A folyamatos mozgást ilyen módon nehéz leírni. Optimálisabb megoldást szükséges alkalmazni.

A mozgó objektumok adatbázisaira vonatkozó felhasználói lekérdezések két kategóriába sorolhatók: múlt-idejű és jövő-idejű lekérdezésekre. Az utóbbiaknak nagy hasznát veszik pl. az időjárás előrejelzéseknél illetve a légi közlekedés felügyeletében. Az ilyen típusú lekérdezések támogatásához általában az objektumok aktuális pozícióját és a sebességet lineáris függvényként tároljuk el az adatbázisban. A továbbiakban a hangsúlyt a jövőbeli lekérdezésekre helyeztem.

A következőkben két alapmegoldást ismertetek a mozgó objektumok indexelésére.

6.1 TPR-fa

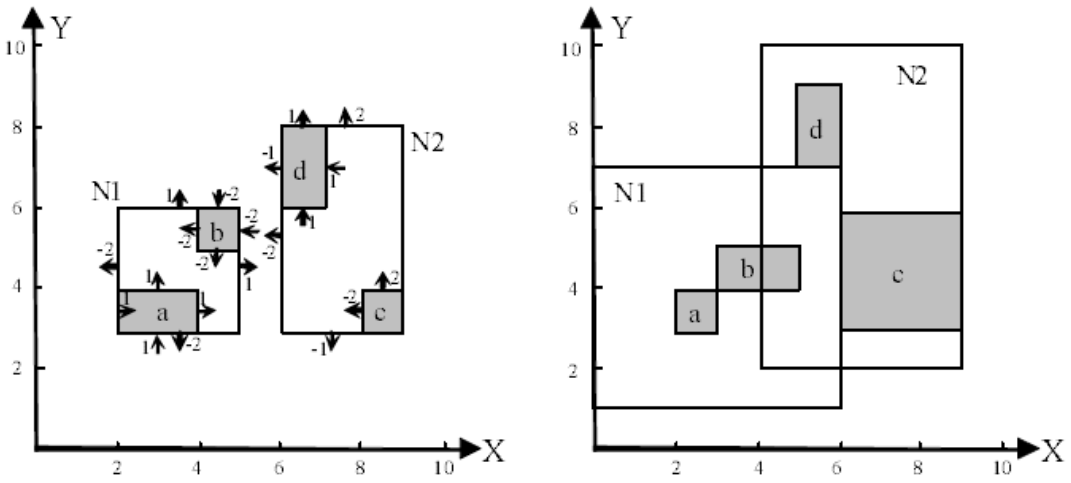
Az R-fáknak több változatuk is van és mindegyik a többdimenziós adatokat a legkisebb befoglaló téglalapjuk alapján kezeli. Ebben a fejezetben azt a típust mutatom be, amely a lineárisan mozgó objektumok R*-fája. Az R-fáknak ezt a típusát TPR-fának (*Time-Parametrized R-tree*) nevezzük. Ennek egyik változata, a TPR*-fa, a legszélesebb körben használt indexstruktúra, amely a mozgó objektumok jövőbeni pozíciójának jóslására alkalmas és felhasználható a jövő-idejű lekérdezésekhez. Ilyen jövő-idejű

lekérdezés eredménye lehet pl. azoknak a repülőknek a halmaza, amelyek a következő 1 órában a viharzónában lesznek (részletesebben lásd 6.3 fejezet). A TPR*-fa megismeréséhez a TPR-fából kell kiindulni, ezért ez a típus kapja ebben a fejezetben a főszerepet. A TPR*-fa ennek segítségével és a műveletek bizonyos módosításaival könnyen előállítható.

A TPR-fa az R*-fának olyan kiterjesztése, amely megadja mozgó objektumok jövőbeni pozícióját, ha adott az objektumok helye és sebessége egy adott időpontban. A struktúra a legkisebb befoglaló téglalap helyett a konzervatív befoglaló téglalapnak nevezett alakzatot használja (*cbr – Conservative Bounding Rectangle*). A *cbr* előáll egy olyan *mbr*-ből, amely egy adott időpillanatban az összes mozgó objektumot lefedő területet reprezentálja illetve egy *vbr*-ből (*vbr – Velocity Bounding Rectangle*), amely az *mbr*-ben, az egyes tengelyek irányában, maximális és minimális sebességgel mozgó objektumok sebesség érékeiből áll elő. Tehát egy *o* mozgó objektum reprezentálása a következőképpen történik:

(1) egy o_R-rel jelölt minimális befoglaló téglalap, amelynek kiterjedése a 0. referencia időpontbeli állapotokat jelöli.

(2) egy o_v-vel jelölt téglalap, amelyre: $o_v = \{o_{v1-}, o_{v1+}, o_{v2-}, o_{v2+}\}$, ahol o_{vi-} (o_{vi+}) az alsó (felső) sebesség korlátja o_v-nek az i-edik dimenzió mentén ($i \in [1..2]$).



6.1 ábra: mozgó objektumok *mbr*-je és *vbr*-je a 0. majd az 1. referencia időpillanatban

Az 5.1 ábrán látható *a, b, c* és *d* objektumok minimális befoglaló téglalapjai láthatók, a rajtuk látható nyilak és a számok a mozgásuk irányát és sebességét jelzik az

egyes tengelyek mentén. Pl. a *c* objektum *vbr*-je: c_v = {-2, 0, 0, 2}, ahol az első két szám tartozik az *x* tengelyhez. Mivel csak két irányban van mozgás, ezért az objektum mérete megnő. Az *a*, *b* és *d* objektumok esetén viszont valós elmozdulás áll elő. Az ábráról leolvasható az is, hogy az elmozdult, következő időpillanatbeli objektumokat reprezentáló *mbb*-k befoglaló téglalapjának (N_1 és N_2) mérete megnőtt és a továbbiakban már nem a legkisebb méretű lesz. Az eredeti *mbr* minden megfelelő sarokpontja a sebességeknek megfelelő irányban és mértékben mozdult el. Mivel vannak olyan sarkok, amelyek fixen maradnak, vagy ellenkező irányba mozdulnak el, ezért a téglalapok mérete megnőhet.

Ha a befoglaló *cbr* (*mbr* + *vbr*) téglalapok fölé egy R*-fát építünk az R-fáknál leírt módon és az R*-fáknál alkalmazott beszúró algoritmussal, akkor előáll a mozgó objektumok TPR-fája. A fa leveleiben az objektumok legkisebb befoglaló téglalapja (*mbb*) kerül, kiegészítve a sebességek téglalapjával. A belső csúcsokban pedig az *mbr*-ek és *vbr*-ek találhatók, amelyek a gyerekcsúcsaikban lévő *mbr*-ek és *vbr*-ek befoglaló téglalapjai lesznek (6.1 ábra). A 0. időpontban építünk egy TPR-fát, és innen a pozíciók és sebességek ismeretében előállíthatók a jövőbeni időpillanatok TPR-fái. A csúcsokat reprezentáló téglalapok a meghatározása a 0. időpillanatban ugyanúgy történik, mint az R-fák esetén. A további időpillanatokban viszont, ahogy azt az ábrán is láttuk, a belső csúcsokban szereplő téglalapok nem feltétlen lesznek a legszűkebbek, de biztosan magukba foglalják a részfájukban szereplő összes alakzatot.

Ha visszatérünk a példámban említett repülőgépes lekérdezéshez, akkor egy időintervallumra vonatkozó ablaklekérdezést kell végrehajtani, hiszen arra vagyunk kíváncsiak, hogy egy viharzónában éppen mely repülők szállnak és melyek lesznek még 10 perc múlva is ott. Az ablaklekérdezés végrehajtása ugyanazon algoritmussal történik, mint az R*-fáknál, kivéve, hogy a lekérdezés pillanatában dinamikusan előálló *mbr*-ekkel kell az összehasonlításokat végezni a fa egyes szintjein. A TPR-fák a [T_C, T_C + H] időintervallumba eső időpillanatokra vonatkozó lekérdezésekre vannak optimalizálva, ahol T_C az aktuális frissítési dátum, H (*horizon*) pedig a fa azon paramétere, amely megadja, hogy meddig szükséges a fát előre látnunk.

Az R*-fákba való új objektum beszúrásakor a következő négy paraméter minimalizálását kellett szem előtt tartani: bejegyzés területe, minden téglalap kerülete, egy csúcsban elhelyezkedő téglalapok átfedése, egy *mbb* középpontja és a szülőcsúcsában elhelyezkedő *dr* téglalap középpontjának távolsága. TPR-fákba való beszúráskor az előzőekben említett metrikákat helyettesítsük a megfelelő társaikkal:

(1) egy N bejegyzés területe helyett: $\int_{T_C}^{T_C+H} A(N,t)dt$, ahol $A(N, t)$ az N bejegyzés területe a t időpontban.

(2) egy N bejegyzés kerülete helyett: $\int_{T_C}^{T_C+H} P(N,t)dt$, ahol $P(N, t)$ az N bejegyzés kerülete a t időpontban.

(3) két dr téglalap átfedése helyett: $\int_{T_C}^{T_C+H} OVR(N_1,N_2,t)dt$, ahol $OVR(N_1, N_2, t)$ az N_1 és N_2 közötti átfedés nagysága a t időpontban.

(4) két dr téglalap középpontjának távolsága helyett: $\int_{T_C}^{T_C+H} CDist(N_1,N_2,t)dt$, ahol $CDist(N_1, N_2, t)$ az N_1 és N_2 középpontjának távolsága a t időpontban.

A beszúrás és törlés algoritmusa egyéb tekintetben pontosan ugyanúgy működik, mint ahogy azt az R*-fák esetén is tettük. A korábbiakban említettem, hogy egy időpontból a következőbe lépve felléphet az a tulajdonság, hogy több elemet összefogó *mbr* nem a legszűkebb lesz. Azonban, egy új elem beszúrásakor vagy egy meglévő törlésekor a megfelelő befoglaló téglalap igazodni fog a benne elhelyezkedő téglalapok legszűkebb befoglalójához. Ez természetesen nem okoz plusz költséget a művelet végrehajtásakor, mert a téglalapot mindenképpen be kell olvasni, majd ismét ki kell írni. Amely téglalapokat a beszúrás nem érinti, azok továbbra sem fognak igazodni a tartalmukhoz.

6.2 TPR-fák költség-modellje

Ahogy az R-fáknál, úgy itt is adható egy költség-modell, amely megjósolja a TPR-fa teljesítményét és megadja azokat a tényezőket, amelyeknek a lekérdezés költségében meghatározó a szerepük.

A modell rövid ismertetéséhez a következő jelöléseket vezetjük be egy o mozgó objektum esetén [15]:

(1) egy o_R-rel jelölt *mbr*, amelyre: $o_R = \{o_{R1-}, o_{R1+}, o_{R2-}, o_{R2+}\}$, ahol o_{Ri-} (o_{Ri+}) az alsó (felső) koordináta korlátja o_R-nek az i-edik dimenzió mentén ($i \in [1..2]$).

(2) egy o_v-vel jelölt *vbr*, amelyre: $o_v = \{o_{v1-}, o_{v1+}, o_{v2-}, o_{v2+}\}$, ahol o_{vi-} (o_{vi+}) az alsó (felső) sebesség korlátja o_v-nek az i-edik dimenzió mentén ($i \in [1..2]$).

(3) $o_{Ri} = [o_{Ri-}, o_{Ri+}]$, amely o terjedelme az i-edik térbeli tengely mentén és $|o_{Ri}| = o_{Ri+} - o_{Ri-}$ a hossza ($i \in [1..2]$)..

(4) $o_{vi} = [o_{vi-}, o_{vi+}]$, amely o terjedelme az i-edik sebesség tengely mentén és $|o_{vi}| = o_{vi+} - o_{vi-}$ a hossza. ($i \in [1..2]$).

(5) az o objektumot *mbr*-je a t időpillanatban a következő módon van reprezentálva: $o_R(t) = o_R + o_v * t$, ahol o_R a 0. referencia időpontbeli téglalap.

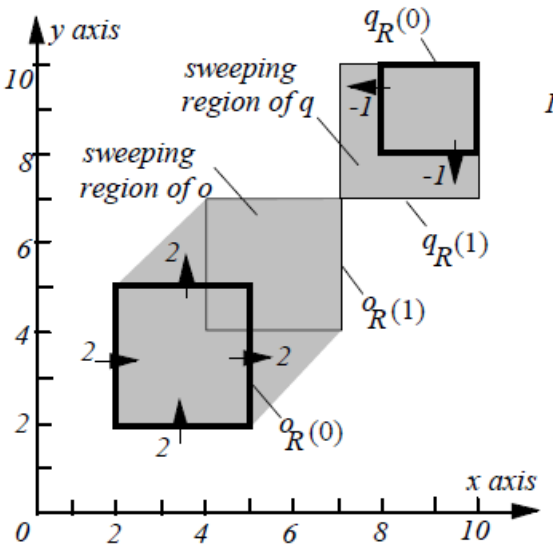
(6) $o_{Ri}(t) = [o_{Ri-}(t), o_{Ri+}(t)]$, amely $o_R(t)$ terjedelme az i-edik térbeli tengely mentén és $|o_{Ri}(t)|$ a hossza ($i \in [1..2]$).

Legyen ezek után o és q egy-egy mozgó objektum a térben. Az o-nak a q-ra vonatkozó transzformált téglalapja o' a következő elemekből áll [15] (6.3 ábra):

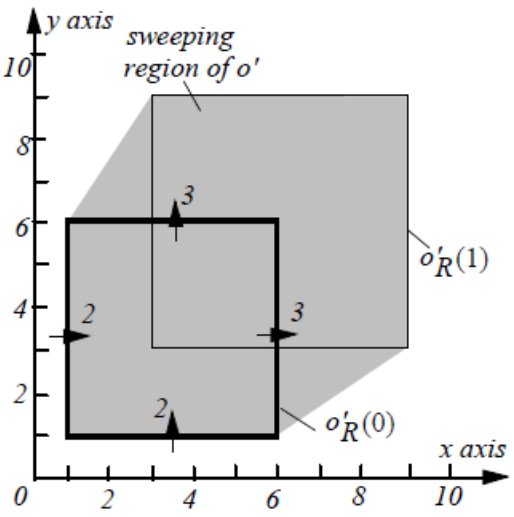
(1) egy $o_{R'}$-vel jelölt *mbr*, amelynek terjedelme az i-edik dimenzió mentén ($i \in [1..2]$): $o_{R'} = [o_{Ri-} - |q_{Ri}| / 2, o_{Ri+} - |q_{Ri}| / 2]$.

(2) egy $o_{v'}$-vel jelölt *vbr*, amelynek terjedelme az i-edik dimenzió mentén ($i \in [1..2]$): $o_{v'} = [o_{vi-} - q_{vi+}, o_{vi+} - q_{vi-}]$.

Adott o mozgó objektum és T időintervallum esetén a mozgástér $SR(o, T)$ az a tartomány, amelyet o megtesz T idő alatt. Ennek területe $A_{SR}(o, T)$ (6.2 ábra).



6.2 ábra: két mozgó objektum (o és q), illetve $SR(o, [0, 1])$ és $SR(q, [0, 1])$



6.3 ábra: transzformált mozgó objektum (o') illetve $SR(o', [0, 1])$

Legyen p egy tartomány lekérdezés, melynek téglalapja egyenletesen tartalmazza a térben elhelyezkedő pontokat. Legyen a kiterjedése az i-edik dimenzió mentén $|p_{Ri}|$, a sebesség vektora p_v, és a lekérdezési időintervalluma $p_T = [p_{T-}, p_{T+}]$. Ekkor a p kérdés megválaszolásához érintett csúcsok átlagos száma $Cost(p) = \sum_{every\ node\ o} A_{SR}(o', p_T)$, ahol o a mozgó téglalap reprezentálása egy csúcsban és o' az o-nak a p-re vonatkozó transzformált téglalapja. Egy o csúcs érintésre kerül egy lekérdezés alkalmával, ha a benne szereplő *mbr* metszi a lekérdezés ablakát a p_T időintervallum alatt. Ez akkor és csak akkor igaz, ha $SR(o', p_T)$ lefedi az lekérdezés ablakának középpontját (statikus pont). Tekintettel, hogy az ablak egyenletesen oszt a pontok terében, ezért a hozzáférés valószínűsége megegyezik az o' téglalap p_T idő alatt létrehozott mozgásterének területével, azaz $A_{SR}(o', p_T)$-vel. Ezt a valószínűséget minden csomópontra összegezve, kapjuk a hozzáférések várható számát.

Mindezekből következik, hogy a fenti paraméterekkel rendelkező lekérdezések akkor lesznek optimálisak, ha az egyenlet minimális értékű lesz.

Említettem, hogy a TPR-fának egy másik típusa az, amelyet a legtöbb térinformatikai rendszer használ mozgó objektumok kezelésére. Ez a típus előállítható a TPR-fából a műveletek továbbfejlesztésével. A TPR-fa és a TPR*-fa szerkezete általánosságban megegyezik. A különbség a felépítésük során adódik. Míg a TPR-fa az R*-fa beszúró és törlő algoritmusait használja úgy, ahogy azok adottak, addig a TPR*-fa ezeknek egy módosított változatát használja, amely próbálja minimalizálni a költségmodellben szereplő egyenlet eredményét és a majdnem optimális alakot elérni. Ezt úgy éri el, hogy közben jobban tükrözi az objektumok mozgását is. Ennek eredményeképpen javítja a módosító műveletek és a lekérdezések teljesítményét a TPR-fákhoz képest.

A TPR-fa egy adott időpillanatban (frissítés időpillanata), a korábbi 4 feltételt figyelembe véve (integrálok), azokat minimalizálva végzi a beszúró és törlő műveleteit. Ezzel szemben a TPR*-fa a mozgó objektum mozgásterét veszi figyelembe módosításkor és azt igyekszik minimalizálni. Ez a mozgástér annak a téglalapnak lesz a kiterjesztése, amely az objektum frissítés után bekövetkező időpillanatbeli állapotához tartozó csomópontnak felel meg. Tehát a TPR*-fa a beszúrás utáni időpillanatban előálló téglalapok közül választja a minimális területűt. A gyökértől a levelekig haladva a csomópontokban a beszúrás előtt elő kell állítani az ahhoz szükséges kiterjesztett téglalapokat és egy elsőbbségi sorban eltárolni. És végül az optimális csúcs a

beszúráshoz az, amelyiknek a kiterjesztett téglalapja a minimális méretű lesz. Ezzel persze a TPR*-fák frissítésének műveletigénye nagyobb lesz, azonban a határoló téglalapok tömörsége nagyban javítja az általános lekérdezések teljesítményét.

Erre az előre gondolkodásra azért van nagy szükség, mert ha pl. egy pontot beszúrunk egy adott pillanatban, akkor előfordulhat, hogy egyetlen téglalapba sem esik bele. Azonban a következő pillanatban az objektumok elmozdulása lehet olyan mértékű, hogy a befoglalóiknak az *mbr*-je annyira megnő, hogy a beszúrt pontunk rögtön beleesik valamelyikbe, vagy akár többe is, ha köztük átfedés jön létre.

6.3 Parametrikus R-fa

Ez az adatszerkezet szintén mozgó objektumok egy hatékony indexelésére alkalmas. Az idő függvényében változó és mozgó téglalapokat kezeli, akár csak az előző struktúra, de attól eltérő a megközelítése. A PR-fa a parametrikus adatmodellt használja fel, amely a mozgó objektumoknak egy természetesebb reprezentálása, mint a több irodalomban is vizsgált mozgó pontoknak. Ez az adatszerkezet az R-fákra épül, annak a beszúró és törlő algoritmusát fejleszti úgy, hogy mozgó alakzatok indexelésére alkalmas legyen.

6.3.1 Parametrikus adatmodell

Tegyük fel, hogy, adott egy mozgó objektumokat tartalmazó adatbázis, amelyben az alakzatok alakja és pozíciója időben folyamatosan változik.

A modell a d dimenziós parametrikus téglalapok leírására szolgál. A modellben egy mozgó objektumot az idő változásával táguló vagy zsugorodó téglalappal adjuk meg, amelyhez $d + 1$ intervallum szükséges. Egy intervallum reprezentálja azt az időtartományt, amelyben az alakzat mozgását nyomon követjük. Ez az intervallum konstans végpontokkal rendelkezik. A többi d darab intervallum az egyes dimenziókhoz megadott időben változó végpontokból állnak.

Legkisebb befoglaló parametrikus téglalap (MBPR): Egy R parametrikus téglalap megfelel egy többdimenziós mozgó téglalapnak, amelyet a modellben a következőképpen definiálhatunk: $R = (I_1, \ldots, I_d, I^{[}, I^{]})$, ahol d a dimenzió értéke és I_i egy zárt intervallum az i-edik dimenzióban ($i \in [1, d]$). Egy I_i-ről a következőket mondhatjuk el: legyenek $x_i^{[}$ és $x_i^{]}$ a t időben lineáris függvények és $t \in [t^{[}, t^{]}]$, ahol a $t^{[}$ kezdő időpont és a $t^{]}$ záró időpont racionális számok. R projekciója (vetülete) az x_i dimenzióra egy egydimenziós parametrikus intervallum lesz $(I_i, t^{[}, t^{]})$. Ebből az

következik, hogy egy d dimenziós parametrikus téglalap ugyanaz, mint egy poliéder a $d + 1$ dimenzióban.

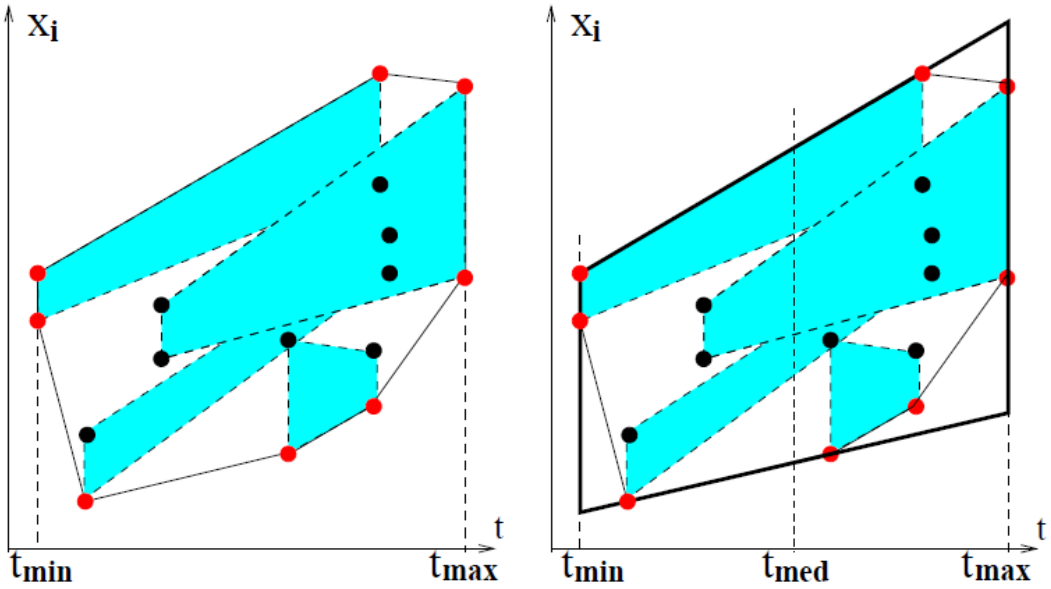
Adott S halmaz, amely a d dimenziós parametrikus téglalapokat tartalmazza. Az S elemeinek minimális befoglaló parametrikus téglalapja R akkor és csak akkor, ha:

(1) R tartalmazza S minden elemét,

(2) R-nek $\forall\ i \in [1, d]$-re az (x_i, t) altérre vett vetületének területe minimális.

Téglalapok meghatározása: Legyen S továbbra is egy d dimenziós parametrikus téglalapokat tartalmazó halmaz és legyen R az S-nek az *MBPR*-je. Ekkor R kezdő / záró időpontja megegyezik az általa befoglalt téglalapok kezdő / záró időpontjainak minimumával / maximumával: $t_{\min} = \min_{r \in S}(r.t^{[}), t_{\max} = \max_{r \in S}(r.t^{]})$. A minimális befoglaló téglalapok megadásának első lépése az időintervallum két végpontjának meghatározása. Ezt egyszerűen megtehetjük a képlet alapján.

Nehezebb feladat R-nek, $\forall\ i \in [1, d]$-re az x_i dimenzióban, az alsó és felső határaihoz tartozó függvényeket meghatározni, azaz az egyes alterekhez tartozó intervallumokat. Mivel S bármely parametrikus téglalapjának leképezése valamely (x_i, t) altérre egy trapézformát ad eredményül négy kitűntetett ponttal (6.3 ábra baloldala), ezért S egészének vetülete valamely (x_i, t) altérre (S_i, $i \in [1, d]$) ezeknek a trapézoknak lesz a befoglaló halmaza $4 * | S |$ számú kitűntetett ponttal.



6.4 ábra: parametrikus téglalapok vetülete (x_i, t) altérre, konvex burok és *MBPR*

Legyen H_i az S_i halmazban lévő trapézoknak a konvex burka. (6.3 ábra baloldala) Ekkor geometriai úton bizonyítható, hogy a konvex burokból a következő két lemma segítségével megadható R legkisebb befoglaló parametrikus téglalap az (x_i, t) térben [4]:

Lemma 1. Maradva az előzőekben projekcióval létrehozott S_i halmazoknál, bizonyítható, hogy R csúcspontjai az (x_i, t) altérre vetítve megegyeznek S_i konvex burkának bizonyos csúcspontjaival, azaz az élei biztosan illeszkednek H_i megfelelő éleire.

Lemma 2. Bizonyítható továbbá, hogy R alsó és felső határolója az (x_i, t) altérben a következő módon illeszkedik a H_i konvex birok bizonyos éleire: a két határoló H_i azon két élének lesz a meghosszabbítása, amelyek metszik a t_{med} középvonalat, ahol $t_{med} = \frac{t_{\min} + t_{\max}}{2}$.

A trapéz két oldalsó szára a konvex burok azon éleinek a meghosszabbítása, amelyek illeszkednek a t_{min} és a t_{max} függőleges vonalakra (6.3 ábra jobboldala).

A fenti lépések végrehajtása következményeként megkapjuk minden dimenzióra R meghatározásához szükséges intervallumokat. A minimális parametrikus téglalap $O(d * n * \log n)$ időben előállítható, ahol n a d dimenziós parametrikus téglalapok száma, amelyeket R magába foglal.

6.3.2 Parametrikus R-fa

A fa segítségével olyan mozgó objektumokra vonatkozó kérdéseket tudunk megválaszolni, mint pl. „Adjuk meg az összes téglalapot a térben, amelyek az R mozgó téglalapot metszik.” (Itt az R-et továbbra is parametrikus téglalapnak tekintjük.). Ez az indexstruktúra teszi lehetővé az adatbázis számára a dinamizmust, azaz, hogy tudjunk objektumokat törölni és újakat beszúrni.

A PR-fa az R-fákhoz hasonlóan, téglalapokat kezel a csúcsaiban, a különbség az, hogy jelen esetben a parametrikus téglalapok kapják a szerepet. A csúcsokban, ezekből a téglalapokból a minimális méretűek kerülnek eltárolásra, fenntartva továbbra is azt a feltételt, hogy a fa kiegyensúlyozott és minden csúcs gyerekeinek a száma $M / 2$ és M között van, ahol M konstans és a lap méretétől függ.

Keresések [4]: Kereséskor kétféle kérdést vizsgálhatunk meg. Az egyik az, hogy két téglalapnak van-e bármely időpillanatban metszete. Vegyük ehhez a két téglalap

projekcióját minden x_i dimenzióban ($i \in [1, d]$) és ellenőrizzük, hogy van-e olyan t_1 időpont, amelyben $\forall\ i \in [1, d]$ -re a két projekciónak van metszete. Ehhez minden dimenzióban előállítjuk azokat az intervallumokat, amikor a két projekcióval létrejött trapéz metszésben van, majd megvizsgáljuk, hogy ezek közül az eredmények közül melyek azok, amelyek mind a d darab dimenzió esetén szerepelnek, azaz megvizsgáljuk, hogy nem üres-e a kapott eredmények metszete. Ha ez a metszet nem üres, akkor a két parametrikus téglalap metszésben van, egyébként nem. Ennek ellenőrzése $O(d)$ időben történik, ahol d a dimenziót jelöli.

A másik kereséssel kapcsolatos kérdés arra ad választ, hogy egy adott objektumhoz tartozó R parametrikus téglalapot mely parametrikus téglalapok metszik. Ekkor a gyökértől a levelek irányába haladunk és minden olyan ágon megy tovább a keresés, ahol átfedés van. Optimális esetben szintenként csak egy csúcs kerül kiválasztásra, ekkor a műveletigény logaritmikus, egyébként ismét $O(d)$, ahol d a dimenzió.

Beszúrás [4]: Új téglalap beszúrása teljesen hasonló módon történik, mint R-fák esetében kiterjesztve az ott szerepelt algoritmust a parametrikus téglalapokra. A gyökértől a levelek felé haladva, minden szinten arról döntünk, hogy melyik részfába szúrjuk be az új parametrikus téglalapot. A beszúrás eredményezhet túlcsordulást, ekkor csúcsvágást kell alkalmazni, és ez kihathat magasabb szintekre is, legrosszabb esetben akár a gyökérig is felgyűrűzhet.

Ahogy a keresésnél láttuk, a műveletigényt az befolyásolja, hogy egy adott szinten hány gyereket szükséges érinteni az keresés folyamán. A cél tehát, hogy a gyerekek számát tartsuk minél alacsonyabban. Ez azt jelenti, hogy az *MBPR*-ek mindig a lehető legkisebbek legyenek. Ennek következménye, hogy egy téglalap beszúrása esetén annak a gyerekcsúcsnak az irányában haladunk tovább, amelyhez tartozó részfa *MBPR*-jét a legkisebb mértékben kell növelni ahhoz, hogy tartalmazza az új parametrikus téglalapot. Egy d dimenziós R parametrikus téglalap méretét a térfogatával lehet jellemezni, ennek kiszámítása a következő képlettel történik:

$$V(R) = \int_{t^{[}}^{t^{]}} \prod_{i=1}^{d} (x_i^{\,]} - x_i^{\,[})dt$$

Tegyük fel, hogy egy új parametrikus téglalapot, R-et, szeretnénk beszúrni egy E belső csúcshoz tartozó *MBPR*-be. Legyenek E_1, ..., E_p ennek a csúcsnak a gyermekei, és R_j az E_j *MBPR*-je ($j \in [1, p]$). Annak a gyerekcsúcsnak a kiválasztása, amelyet a

legkisebb mértékben kell növelni ahhoz, hogy tartalmazza *R*-et, a következő képlet segítségével tehető meg:

$$\min_j Enl\arg e(R_j, R) = \min_j (V(FindMBPR(R_j, R)) - V(R_j))$$

Ha egy telített, már *M* bejegyzést tartalmazó csúcshoz kell beszúrni egy újabb téglalapot, akkor a művelet elvégzése után túlcsordulást fog bekövetkezni. Ez esetben létrehozunk egy új csúcsot, és az *M* + 1 bejegyzést a két csúcs között szétosztjuk. A két csoport különválasztásához megkeressük azt a két bejegyzést, amelyek közös *MBPR*-je a legnagyobb térfogattal rendelkezik:

$$\max_{r_i, r_j \in E} V(FindMBPR(r_i, r_j)) - (V(r_i) + V(r_j))$$

Ezzel a képlettel megtaláltuk az egyes csoportokat reprezentáló elemeket. Ezt a két bejegyzést a nagy térfogat miatt, nem ajánlott egy közös csúcsban tárolni. A továbbiakban ezeknek az *MBPR*-jét bővítjük a többi bejegyzés beszúrásával. A beszúrás minden esetben úgy történik, hogy figyelembe vesszük azt, hogy mennyivel növelik a létrejött két csoport *MBPR*-jének térfogatát, majd ahhoz szúrjuk be, amelynél a növekedés minimálisabb lesz.

Egy PR-fába való parametrikus téglalap beszúrásakor minden szinten meg kell találni a megfelelő részfát és annak *MBPR*-jét növelni kell a beszúrandó téglalap méretével. Két parametrikus téglalap *MBPR*-jének előállításához $O(d)$ idő szükséges. Egy parametrikus téglalap térfogatának kiszámítása ismét $O(d)$ időt igényel. Mivel minden csúcsnak maximum *M* számú gyermeke lehet, ezért ha mindegyikre elvégezzük a téglalap növelést, majd kiválasztjuk közülük a minimum térfogatút, akkor annak műveletigénye $O(d * M)$ lesz minden egyes szinten. *n* számú parametrikus téglalapból épített fa magassága $\log_M n$, így a gyökértől a megfelelő levélig eljutni $O(d * M * \log_M n)$ időt vesz igénybe. Ezzel megadtam a beszúrás műveletigényét, amikor csúcsvágást nem hajtunk végre.

Ha még valamely szinteken csúcsvágásra is szükség van, akkor minden vágáskor M^2 -szer elő kell álltani két téglalap *MBPR*-jét, hogy meghatározzuk a két új csúcs egy-egy referencia bejegyzését. Ennek műveletigénye $O(d * M^2)$. Ahhoz, hogy a hátralévő *M* - 2 darab téglalapot elhelyezzük valamely csúcsban, $O(d * M)$ idő szükséges. Ezért a beszúrás teljes műveletigénye $O(d * M^2 * \log_M n)$ lesz, ahol *M* a lapméret (maximális bejegyzések száma egy csúcsban), *d* a dimenzió mértéke és *n* a parametrikus téglalapok

száma. Egy új elem beszúrása után is megmarad a fa kiegyensúlyozottsága, hiszen a csúcsvágásokat ennek megfelelően végeztük.

Törlés [4]: Az objektum azonosítókat erősen ajánlott egy másik keresőfában is eltárolni, úgy hogy egy pointer mutasson a valódi tárolás helyére. Ennek következménye, hogy minden módosításkor ezt a fát is frissíteni kell. A keresőfa helyett használhatunk hasító-táblát is. Törlésénél először ebben a fában keressük ki az objektum azonosítóját, így gyorsabban megkapjuk az objektum helyét a PR-fában. A másodlagos fából való törlés műveletigénye $O(\log_M n)$. Ezután töröljük a PR-fából is a hozzá tartozó bejegyzést. A PR-fában természetesen újra kell számolni azt az *MBPR*-t, amely a törölt csúcsot tartalmazta, és ezt el kell végezni a fában felfelé haladva, amíg már nincs szükség további csökkentésre. Az újraszámolások műveletigénye $O(d * \log_M n)$. Azt is figyelembe kell venni, hogy egy csúcsnak legalább $M/2$ bejegyzést kell tartalmaznia, és ennek a feltételnek a törlés után is fenn kell állnia. Ha ez mégsem áll fenn, akkor a csúcsot meg kell szüntetni és a bejegyzéseit újra beszúrásokkal szét kell osztani a vele egy szinten elhelyezkedő csúcsok között, amely $O(d * M^3 * \log_M n)$ időben végezhető el. Ez a vizsgálat akár a gyökérig is felgyűrűzhet.

Mindezekből következik, hogy egy törlés teljes műveletigénye $O(d * M^3 * \log^2_M n)$ lesz.

6.4 Gyakorlati felhasználás

A jövő-idejű lekérdezéseknek két csoportja között tehetünk különbséget, a várt eredménytől függően. Az egyik csoportot alkotják a jövő-idejű intervallum lekérdezések. A másik osztályba tartoznak a jövő-idejű paraméteres lekérdezések. Mindkettő eredménye két tábla térbeli összekapcsolásán alapul. Általában, a térbeli összekapcsolások két térbeli objektumok halmaza között, valamilyen térbeli állítás alapján létrejövő kapcsolatot jelent (pl. átfedés).

A két összekapcsolt objektumhalmaz lehet, hogy más-más indexelést használ, sőt akár az is lehet, hogy az adattárolási módjuk is különbözik. Ha a korábbi repülőgépes példához visszatérek, akkor mondható pl. az, hogy tároljuk el a repülőket, mint vektoros adatokat, majd mozgó objektumok lévén, indexeljük őket egy TPR*-fával. A viharzóna pedig legyen egy szürke, raszteres adattárolással létrejött terület, amelyet indexelhetünk pl. négyfa típusú adatszerkezettel. Ekkor a két tábla térbeli feltételen alapuló

összekapcsolását igénylő, korábban is feltett jövő-idejű kérdés lehet, az, hogy „Melyek azok a repülők, amelyek 10 percig a viharzónában lesznek?”. A gyakorlati alkalmazások esetén sokszor fordul elő ilyen eset, ezért tartottam fontosnak néhány szólt szólni róluk.

Visszatérve a lekérdezések két csoportjához, a példát kicsit részletesebben véve a következők mondhatók: legyen adott 6 repülőgép a mozgásukat ábrázoló vektorokkal, illetve egy viharzóna raszteresen ábrázolva. A *B* repülő a 0. referencia időpontban éppen a viharzónában van. A 10. ilyen időpontban már éppen kilépett, miközben *D* és *E* meg berepült. Legyen a referencia időpontok közötti eltelt idő 1 perc. Ekkor az első csoportot alkotó jövő-idejű intervallum lekérdezésre példa a következő: „Melyek azok a repülők, amelyek a következő 9 percben és 59 másodpercben a viharzónában lesznek?” Ennek eredménye a *B* repülő lesz. Egy másik lekérdezés lehet, hogy „Adjuk meg az összes repülőt, amelyek éppen most vannak a viharzónában, adjuk meg azt a pillanatot, amikor a jelenlegi helyzet megváltozik és azt az eseményt, amely miatt a változás bekövetkezik”. Az ilyen típusú lekérdezéseket lehet a második, jövő-idejű paraméteres lekérdezések, osztályába sorolni. Erre a válasz pedig: jelenleg a *B* repülő van a viharzónában és 10 perc múlva változik a helyzet, mert akkor *B* kilép, *D* és *E* pedig belépnek.

Látható, hogy mindkét lekérdezés a jövőre vonatkozik, a különbséget az időintervallum és az időpillanat fogalma okozza (feltételezve, hogy a hajók irányvektora nem változik meg két egymást követő pozíciófrissítés között). Azonban a különböző alkalmazásoktól függően, hogy milyen gyakoriságúak az időbeli mintavételezéseink, azaz az objektumok pozíciójának frissítései és az irányvektorok hirtelen változásai, közel pontos eredményeket kaphatunk. Figyelembe kell venni azt is, hogy jelen esetben a repülők mozgásához képest a viharzóna fix (statikus), de alapjában véve az is lehet egy mozgó objektum.

Ebben a rövid alfejezetben felvetett valós élet problémáira nincs adva egy egységes eljárás vagy módszer, amely a szakirodalmakban le lenne írva. A konkrét gyakorlati alkalmazásoknál dől el, hogy melyik esetben milyen tárolási módot és milyen indexelési technikát használjunk.

7. Összefoglalás

Dolgozatomban áttekintettem a térinformatikai indexelési módszerek leggyakrabban használt változatait. Két területen mélyebb irodalomkutatást és önálló elemző meggondolásokat végeztem. Az egyik önállónak tekinthető részben a konvex alakzatok indexelésével foglalkoztam. A másik mélyebben megvizsgált – tananyagban nem szereplő – kérdés a mozgó objektumok indexelése volt.

A dolgozat megírásának elsődleges hozadéka számomra az volt, hogy három érintkező tudományág szempontjait egyidejűleg kellett, és talán sikerült egyeztetni a tárgyalásban. Az adattárolás és információkezelés már az ügyviteli alkalmazások során is felveti a hatékony indexelés kérdését. A térinformatikai alkalmazások azonban sokkal nagyobb súlyt adnak ennek a kérdésnek, és jóval nagyobb változatosságot igényelnek a megoldásban. A megfelelő indexelési eljárások megértése, illetve önálló kidolgozása csak az algoritmusok és adatstruktúrák témakör mélyebb ismeretével lehetséges.

Alapvető felismerés volt dolgozatom elkészítése során, hogy eltérő kezelési módszert kíván az egy-, kettő-, illetve háromdimenziós térinformatikai környezet, és további változatosság jelenik meg a mozgó objektumok tanulmányozásakor. Diplomamunkám a teljesség igénye nélkül – különböző változataikkal együtt – több mint 20 indexelési módszert dolgoztam fel. Az indexelési eljárások között a dimenzió számon túl finom különbséget tesznek azok a feladatok, amelyekre hatékonyan alkalmazzák őket.

Dolgozatom elkészítésébe alapos munkát fektettem. A befejezést izgalmassá és emlékezetessé tette az a körülmény, amikor váratlan nehézségbe ütköztem a trapézokra szeletelt konvex alakzatok indexelésének kérdésében. Úgy érzem, hogy a felmerült problémára adott megoldásom esetleg nem a legtökéletesebb, de a választott témakör mélyebb megértését bizonyítja.

Kronológia

Adaptív kd-fa: J. H. Friedman, J. L. Bentley, R. A. Finkel, *1979*.

B-fa: Edward M. McCreight, Rudolf Bayer, *1972*.

BSP-fa: Bruce F. Naylor, Henry Fuchs, Z. M. Kedem, *1980*.

Fix grid: J. H. Friedman, J. L. Bentley, *1979*.

Grid file: H. Hinterger, J. Nievergelt, K. C. Sevcik, *1984*.

Hilbert-görbe: David Hilbert, *1891*.

kd-B-fa: John T. Robinson, *1981*.

kd-fa: J. L. Bentley, *1975*.

Pakolt R-fa: Daniel Leifker, Nick Roussopoulos, *1985*.

Pont négyfa: J. L. Bentley, R. A. Finkel, *1974*.

Ponttartomány négyfa: Hanan Samet, *1990*.

PR-fa: Mengchu Cai, Peret Revesz, *2000*

R*-fa: B. Seeger, H. P. Kriegel, N. Beckman, R. Schneider, *1990*.

R^+-fa: C. Faloutsos, N. Roussopoulos, T. Sellis, *1987*.

R-fa: Antonin Guttman, *1984*.

Szegmensfa: J. L. Bentley, *1977*.

Tartomány négyfa: Hanan Samet, *1990*.

TPR-fa: C. S. Jensen, Mario A. Lopez, Scott T. Leutenegger, Simonas ˇSaltenis, *2000*.

TPR*-fa: Yufei Tao, Dimitris Papadias, Jimeng Sun, *2003*.

Z-rendező fa: J. A. Orenstein, T. H. Merrett, *1984*.

Z-vonal: G. M. Morton, *1966*.

Ábrajegyzék

2. Fejezet

3. Fejezet

4. Fejezet

5. Fejezet

6. Fejezet

Irodalomjegyzék:

[1] Philippe Rigaux, Michael O. Scholl, Agnes Voisard: *Spatial Databases: Width Application to GIS*, Morgan Kaufmann Publishers, 2002., [410], ISBN 1 55860 588 6

[2] Hanan Samet: *The Design and Analysis of Spatial Data Structures*, Addison-Wesley Publishing Company, Inc. 1990., [493], ISBN 0 201 50255 0

[3] Hector Garcia-Molina, Jeffrey D. Ullman, Jennifer Widom: *Adatbázisrendszerek megvalósítása*, Panem Könyvkiadó Kft., Budapest, 2001., [682], ISBN 963 545 280 2

[4] Peter Z. Revesz: *Introduction to Constraint Databases*, Springer, 2002., [393], ISBN 0 387 98729 0

[5] Elek István: *Bevezetés a geoinformatikába*, ELTE Eötvös Kiadó, Budapest, 2006., [365], ISBN 963 463 864 3

[6] Hee-Kap Ahn, Nikos Mamoulis, Ho Min Wong: *A Survey on Multidimensional Access Methods*, 2001. http://www.cs.uu.nl/research/techreps/repo/CS-2001/2001-14.pdf

[7] Volker Gaede, Oliver Günther: *Multidimensional Access Methods*, 1998. http://mistral.in.tum.de/rwork/gg98.pdf

[8] Hanan Samet: *Spatial Data Structures*, 1995. http://www.liacs.nl/~nikolov/StudSem/DS_sds.pdf

[9] Peter van Oosterom: *Spatial Access Methods*, 1999. http://www.gdmc.nl/oosterom/bb2short.pdf

[10] Dr. Katona Endre: T É R I N F O R M A T I K A (Előadási jegyzet Programtervező matematikus és geoinformatikus hallgatók számára, SZTE, 2006) http://www.inf.u-szeged.hu/~katona/gis.pdf

[11] Jeffrey D. Ullman, Jennifer Widom: *Adatbázisrendszerek alapvetés*, Panem Könyvkiadó Kft., Budapest, 1998, [507], ISBN 978 9 635454 91 4

[12] Atallah, M. J. (ed.): *Algorithms and Theory of Computation Handbook*, CRC Presss, Boca Raton, 1999, [1218], ISBN 0 8493 2649 4

[13] Z-order (curve), 2011.
http://en.wikipedia.org/wiki/Z-order_(curve)

[14] Hilbert curve, 2011.
http://en.wikipedia.org/wiki/Hilbert_curve

[15] Antonio Corral, Manuel Torres, Michael Vassilakopoulos, Yannis Manolopoulos: *Predictive Join Processing between Regions and Moving Objects*, 2008.
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.15.3545&rep=rep1&type=pdf

Köszönetnyilvánítás

Szeretnék köszönetet mondani **Dr. Benczúr András** professzor úrnak azért, hogy elvállalta a témavezetésemet. Köszönöm türelmét és hasznos tanácsait, azt, hogy a rengeteg elfoglaltsága ellenére is időt tudott szakítani konzultációs időpontokra.

Kiemelten szeretnék még köszönetet mondani **Dr. Fekete Istvánnak**, akihez bármikor fordulhattam az algoritmusok és az adatstruktúrák területén felmerült kérdéseimmel, és aki mindig feltétel nélkül állt rendelkezésemre. Segítsége és tanácsai nélkül ez a dolgozat nem sikerült volna ilyen részletesre.