Quadtrees

- Structuri de date ierarhice (arbori) – au la bază principiul de descompunere recursivă (asemănătoare cu divide e impera)
- Denumirea provine de la faptul că în general este vorba despre arbori în care fiecare nod are (cel mult) 4 descendenţi.
- Diferenţiate după:
    1. Tipul de date pe care îl reprezintă: point data, regiuni, curbe, suprafeţe, volume
    2. Principiul de descompunere: descompunere în părţi egale la fiecare nivel (eg. poligoane regulate) sau descompunere pe baza anumitor condiţii de intrare
    3. Rezoluţia descompunerii: poate fi fixată dinainte sau poate depinde de anumite proprietăţi ale datelor de intrare

1. Quadtree pentru regiuni

Exemplu: quadtree pentru regiuni – descompunerea unei imagini binare în regiuni uniforme

Observaţie:

- pentru a obţine blocuri maximale omogene – reuniunea de blocuri adiacente pătrate uniforme (etapa de merging)

- deşi structura pătrată a unei regiuni este cea mai simplă există şi alte tipuri de descompunere: triunghiuri echilaterale, triunghiuri dreptunghice isoscele, hexagoane echilaterale

- cresc viteza de execuţie a anumitor algoritmi pe imagini, suprafeţe, structuri geometrice, dat fiind că majoritatea acestor algoritmi se reduc la o parcurgere, de obicei în preordine, a quadtree-ului asociat.

Determinarea vecinilor

Două regiuni se numesc vecine, dacă sunt adiacente.

Observaţie: adiacenţa în spaţiu nu înseamnă neapărat o relaţie simplă în cadrul nodurilor corespunzătoare din arbore.

Exemplu: - regiunile x şi y sunt adiacente, dar în cadrul quadtree-ului nu sunt nici în relaţia de părinte-fiu, nici în relaţia de frate-frate. Se pune deci problema găsirii în arbore a două noduri, care în spaţiu reprezintă regiuni vecine.

Adiacenţă şi vecinătate:

Fiecare nod al unui quadtree corespunde unei regiuni/unui bloc din imagine. Fiecare bloc are 4 laturi şi 4 colţuri.

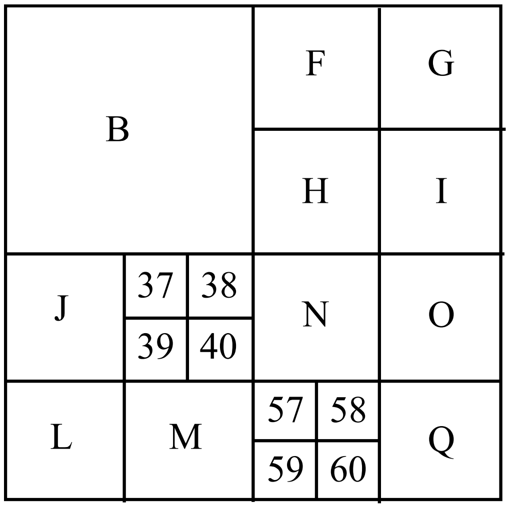
Notaţii:

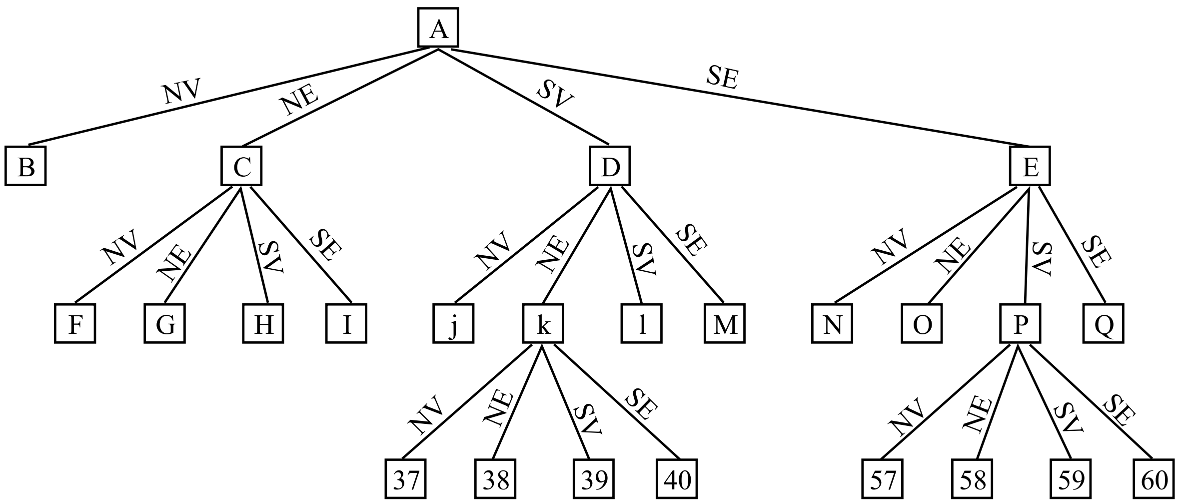
- cele patru laturi ale unui bloc prin N(nord), S(sud), E(est), V(vest).
- cele patru colţuri: NV, NE, SV, SE

Adiacenţă: spunem că două blocuri P şi Q disjuncte sunt adiacente după direcţia D (D=N,S,E,V, NV,NE,SV,SE), dacă

- P are o parte din latura de pe direcţia D comună cu Q – de exemplu în figură blocul F este vecin la E cu blocul G.
- Coluţul din direcţia D al blocului P este adiacent cu colţul opus al blocului Q – în exemplu, blocul 38 este NE adiacent cu blocul H

Exemplu:





Observaţii:

- adiacenţa este definită atât pentru blocuri reprezentate prin frunze în arbore cât şi pentru blocuri reprezentate prin noduri interne.

- Un bloc P poate avea după o anumită direcţie mai mulţi vecini – în exemplu blocul B are după direcţia E vecinii: F şi H.

Probleme:

(1) Determinarea vecinilor care sunt adiacenţi cu îngtreaga latură a unui anumit bloc – de exemplu: . În acest caz: pentru P determinarea celui mai mic bloc cu latura mai mare sau egală cu cea a lui P şi care este adiacent cu P după direcţia D.

(2) Determinarea acelor vecini care sunt adiacenţi numai cu un segment din latura unui anumit bloc – de exemplu: . În acest caz vecinul căutat trebuie specificat prin informaţii suplimentare

În continuare prezentăm tipurile de cereri care pot fi efectuate asupra unui quad-tree pentru determinarea de vecini:

Notaţii:

G = „greater then or equal”

C = „corner”

S = ”side”

N = „neighbor”

(1) GSN(P,D) = cel mai mic bloc adiacent cu latura din direcţia D a blocului P şi care are latura mai mare sau egală cu latura lui P.

Exemplu: GSN(J,N) = B; GSN(N,V) = K (care se descompune în blocurile 37,38,39,40).

(2) CSN(P,D,C) = cel mai mic bloc care este adiacent cu latura D şi colţul C al nodului P.

Exemplu: CSN(J,E,SE) = 39; CSN(Q,V,NV) = 58.

(3) GCN(P,C) = cel mai mic bloc adiacent cu P şi aflat de partea opusă a colţului C al blocului P şi care are latura mai mare sau egală cu latura lui P. Atenţie: acest bloc trebuie să intersecteze suprafaţa inclusă de colţul opus al clolţului C.

Exemplu: GCN(J,NE) = B NU 37; GCN(B,SE) = E; GCN(58,NV) = N.

(4) CCN(P,C) = cel mai mic bloc aflat de partea opusă a colţului C al blocului P. Şi în acest caz este obligatoriu ca blocul astfel determinat să se intersecteze cu suprafaţa determinată de colţul opus colţului C al lui P.

Exemplu: CCN(58,NV) = N; CCN(59,NE) = 58.

Observaţii:

- GSN, CSN, GCN, CCN nu definesc relaţii 1 – la – 1. Pot exista mai multe blocuri care au acelaşi vecin: GSN(H,E) = B şi GSN(F,E) = B.

- Relaţiile de vecinătate astfel definite nu sunt neapărat simetrice: GSN(H,E) = B, dar GSN(B,V) = C.

- Un bloc care nu se află pe marginea imaginii are minim 5 vecini. Acest lucru poate fi dedus din următoarele observaţii:

  - Un bloc nu poate fi adiacent cu două blocuri de dimensiuni mai mari aflate pe direcţii opuse (vezi fig.)

  - Un bloc poate avea doar doi vecini de latură mai mare, aflaţi pe direcţii care nu sunt opuse (ex: vest şi nord). Pe celelate două laturi şi pe un colţ mai sunt vecini mai mici sau egali ca dimensiune. (vezi fig.).

- Un nod are maxim 8 vecini (de dimensiuni mai mari sau egale).

- În continuare rămân de interes doar realţiile: GSN şi GCN.

Determinarea vecinilor adiacenţi după o direcţie verticală sau orizontală.

Problematică: determinare Q = GSN(P,D)

Idee generală este aceea de a urca de la nodul P către primul strămoş comun cu Q, iar apoi de a coborâ de la acest strămoş comun către Q.

Observaţii:

- Dacă direcţia D = E, căutăm vecinul de la E, deci P se află pe ramuri SV sau NV faţă de strămoşul comun.

- Dacă direcţia este D = V, căutăm vecinul la V deci P se află pe ramuri SE sau NE faţă de strămoşul comun.

- Dacă direcţia D = N, căutăm vecinul de la N, deci P se află pe ramuri SV sau SE faţă de strămoşul comun.

- Dacă direcţia D = S, căutăm vecinul de la S, deci P se află pe ramuri NV sau NE faţă de strămoşul comun.

Stămoşul comun se obţine când se urcă de la nodul curent către părinte pe o ramură ce nu conţine direcţia de căutare!

Exemplu: GSN(N,V) = K. Se observă că se urcă de la N către E, N fiind descendentul NV al lui E. Apoi de la E spre A, E fiind descendentul SE al lui A. În acest moment subarborele care îl conţine pe K se află la EST faţă de nodul curent, care este chiar rădăcina. Deci vecinul de la

VEST va fi pe una dintre ramurile NV sau SV ale lui A. Coborârea în arbore către vecinul căutat se face pe ramuri simetrice faţă de direcţia D relativ la ramurile pe care s-a făcut urcarea către A.

Deci: urcarea s-a făcut pe ramurile NV, SE, iar simetria este D=V, deci ramurile simetrice vor fi SV (simetricul lui SE faţă de verticală) şi NE (simetricul lui NV faţă de verticală) şi se observă că se ajunge la vecinul căutat K.

Algoritm general pentru GSN:

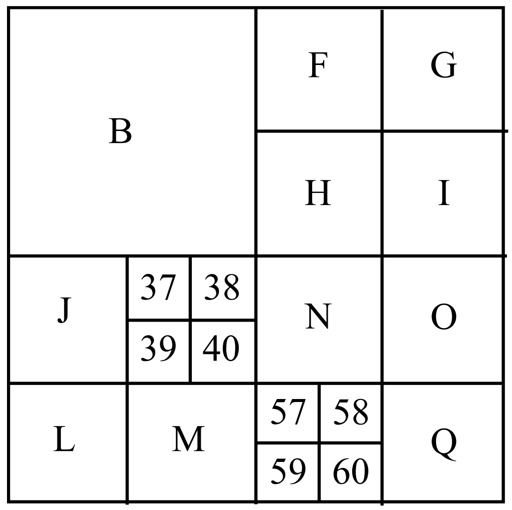
GSN(P,D): - urc de la nodul curent către părinte cât timp nodul curent este unul dintre descendenţii de direcţie D ai părintelui. Apoi cobor de la nodul curent la care am ajuns pe ramuri simetrice faţă de D cu ramurile pe care s-a urcat.

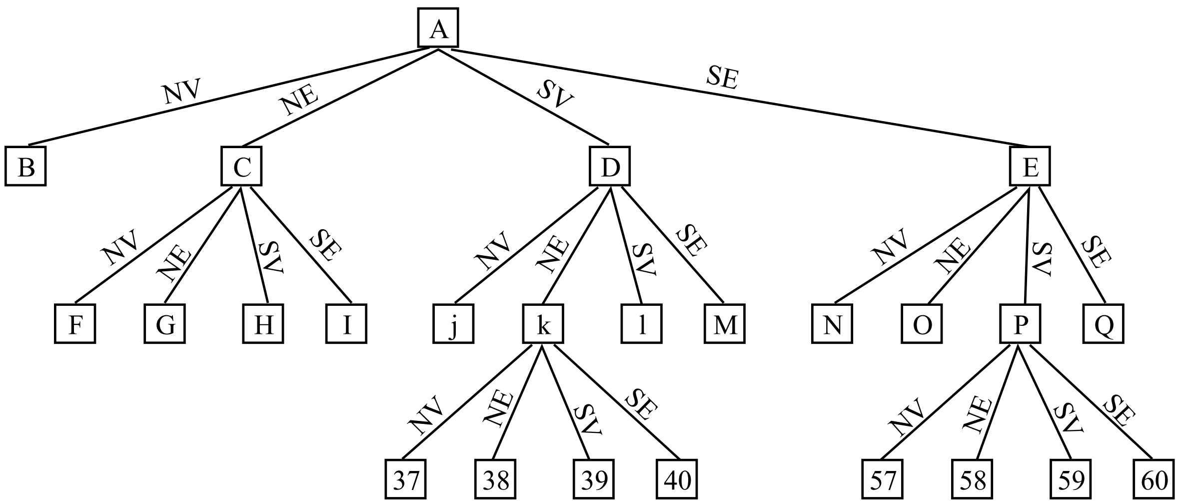
```
GSN(P,D)
nod=P
parinte=nod->p
Stiva=∅
cat timp parinte≠NULL si ramura pe care urc de la nod la parinte
contine D
      • pune pe Stiva simetricul ramurii de urcare de la nod la
        parinte fata de directia D
      • nod=parinte
      • parinte=nod->parinte
sfarsit cat timp
daca parinte=NULL atunci
      RETURN NULL
sfarsit daca

plaseaza pe stiva ramura simetrica fata de D a ramurii
corespunzatoare a lui nod fata de parinte

nod=parinte
cat timp Stiva≠∅
      directie⇐Stiva
      daca nod≠frunza atunci
            nod=nod->directie
      altfel BREAK
sfarsit cat timp
RETURN nod
```

Exemple:





1. GSN(39, N)

Stiva = ∅, nod = (39), parinte = (K), 39 = P->SV, D = N

Deci ramura de urcare nu conţine N => strămoşul comun este chiar K => se oreşte cât timp

Se plasează pe stivă simetricul lui SV faţa de orizontală => S=(NV)

=> cobor din K pe ramura NV şi ajung la nodul 37

Din imagine se vede clar că s-a determinat vecinul dorit.

2. GSN(57,V)

Stiva = ∅, nod = (57), parinte = (P), (57) = (P)->NV, D = V

Deci ramura de urcare conţine direcţia V => plasez pe stivă simetricul lui NV faţă de verticală => Stiva = (NE)

În plus nod = (P), parinte = (E) (P) = (E)->SV, deci ramura de urcare conţine V =>

Stiva =(SE, NE). În plus nod = (E), parinte = (A) şi (E) = (A)->SE. Nu mai conţine direcţia => ne oprim din ciclare.

În plus se pune pe stivă SV, deci Stiva = (SV, SE, NE).

De la rădăcină se coboară pe ramurile din stivă: A->SV = D, D->SE =M dar M este frunză deci ne oprim şi vecinul căutat este M (se verifică corectitudinea în fig.)

Determinarea vecinilor cu funcţia GCN, adică adiacenţi cu un colţ după o anumită direcţie.

GCN(P, D_1D_2): notăm direcţia de adiacenţă cu D_1D_2, unde $D_1 \in \{N,S\}$ şi $D_2 \in \{E,V\}$.

Idee generală: caut un nod strămoş al lui P vecin cu un nod strămoş al nodului căutat. Acest lucru se realizează după cum urmează:

- Se urcă de la nodul curent spre părinte, până când nodul se află pe partea $D_1D_2 \neq D_3D_4$ a părintelui. În acest moment nodul curent devine acest părinte, pe care îl notăm cu Q.
- **Dacă** $D_1 \neq D_3$ şi $D_2 \neq D_4$ atunci Q este strămoş comun şi tot ceea ce trebuie să fac este să cobor din Q pe arce complementare cu cele pe care s-a urcat, adică dacă s-a urcat pe arc SV se coboară pe arc NE.
- **altfel dacă** $D_1 \neq D_3$ atunci caut vecinul R lui Q după direcţia D_2 cu algoritmul GSN(Q,D_2), R va deveni nodul curent.
- **altfel dacă** $D_2 \neq D_4$ atunci caut vecinul R lui Q după direcţia D_1 cu algoritmul GSN(Q,D_1), R va deveni nodul curent.

Din nodul R astfel obţinut se coboară pe arce complementare cu cele pe care s-a ajuns la Q.

Perechile de arce complementare: (NV, SE) (NE, SV).

Exemple:

GCN(57, NV):

- urcăm la părintele P pe ramura NV egală cu direcţia din apel, deci continuăm la părintele lui P, E, pe ramură SV ≠ NV
- Dar D1=N, D2=V şi D3=S, D=V, deci D1≠D3 şi D2=D4 Q=E, rezultă căutarea vecinului R al lui Q, după algoritmul GSN(E, V), adică urc la părinte, cât timp sunt pe o ramură cu S şi nu am ajuns la rădăcină. Urc de la E la A pe ramura SE

- Părintele este A = rădăcina. Acum se coboară confrom algoritmului GSN pe ramura SV, adică la D, care este vecinul la vest al lui E
- Acum, conform algoritmului GCN, cobor pe ramură opusă a lui SV, adică NE şi ajung la K şi de aici cobor pe ramura SE şi ajung la 40
- Din figură se observă că am ajuns exact la vecinul căutat

GCN(38, NE):

- Urcăm la părintele K pe o ramură NE, egală cu direcţia de apel, deci se continuă urcarea la părintele lui K, D, pe o ramură NE, tot egală cu ramura de apel. Se urcă la părintele lui D, care este rădăcina A, pe o ramură SV≠NE, cu D1=N, D2=E, D3=S, D4=V.
- Deci D1≠D3 şi D2≠D4. Am ajuns la un părinte comun al lui 38 cu vecinul căutat. Se coboară acum pe ramurile NE la C şi SV la H. Nu mai există descendenţi pentru H, deci algoritmul se opreşte.
- S-a obţinut vecinul H, care este cel căutat (vezi fig,).

GCN(N, NE):

- Urc de la N la părintele E pe o ramură NV. Observăm NV≠NE cu D1=N=D3, D2=E, D4=V.
- Deci se aplică GSN(E, N). Urcăm de la E la părintele A (rădăcina) pe o ramură SE. De aici se coboară pe o ramură simetrică faşă de orizontală, deci pe ramura NE la nodul C.
- Acum s-a găsit vecinul la N al lui E, care este C.
- De aici se continuă coborârea conform algoritmului GCN pe o ramură complementară cu NV, adcă SE şi se ajunge la I.
- Vecinul căutat este deci I, ceea ce rezută şi din figură.

Aplicaţii: algoritmul Split and Merge de segmentare pentru imagini digitale.

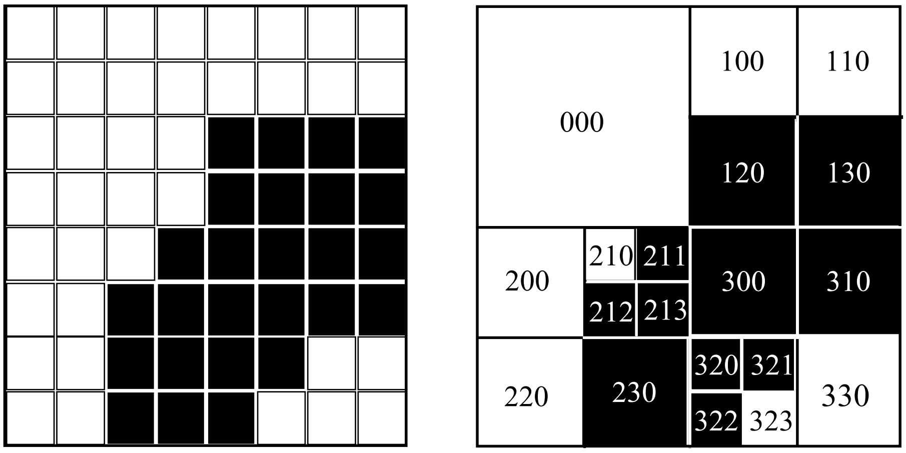
Implementare – Linear Quadtrees

Principala problemă a implementării clasice a unui arbore quad, adică folosind o structură de pointeri, în cazul unei imagini, este numărul mare de noduri interne necesare. De fapt în aplicaţii reale, în cazul imaginilor se utilizează o structură de date numită arbori quad liniari – Linear Quadtrees – care reprezintă de fapt lista frunzelor în parcurgerea de la stânga la dreapta. Fiecare frunză este o structură de tip nod, care conţine un câmp pentru culoare, un câmp pentru nivelul la care se află în arbore şi un cod unic în baza 4.

Codul fiecărei frunze este alcătuit prin interclasarea codurilor binare ale coordonatelor y şi x ale pixelului din colţul stânga-sus al blocului din imagine reprezentat prin frunză. Codul binar obţinut prin iterclasarea celor două coduri este transformat într-un cod în baza 4 prin gruparea cifrelor binare două câte două într-o cifră din baza 4.

Algoritmul presupune imagini de dimensiuni $2^n \times 2^n$.

În figura de mai jos este prezentat modul de codificare al frunzelor obţinute prin descompunerea unei imagini binare $2^3 \times 2^3$ în blocuri de culoare uniformă.



Arborele corespunzător este:

Iar în formă liniară:

Modul de construcţie a codurilor corespunzătoare blocurilor este:

NU SE PREZINTA LA CURS

Algoritmul de construcţie al unui arbore quad liniar.

Se consideră o listă de blocuri active ActiveNodes. Nodurile active reprezintă blocuri de pixeli, care nu au fost încă analizate în totalitate, deci nu se ştie încă, dacă mai trebuie deivizate sau nu.

Fiecare bloc este reprezentat printr-un nod cu următoarele câmpuri:

- Culoare – *color* = culoarea primului pixel al blocului.
- Adâncime – *depth* = dată de gradul de descompunere al frunzei. Pentru rădăcină atributul depth = n (unde dimensiunea imaginii este $2^n \times 2^n$). În general pentru un nod oarecare atributul depth = k, unde dimensiunea blocului reprezentat de către nodul respectiv este $2^k \times 2^k$.
- Pointer la nodul părinte.
- Cod = codul corespunzător blocului respectiv, calculat în maniera prezentată anterior.

Iniţial lista de noduri active conţine doar nodul reprezentând blocul format din toată imaginea, care reprezintă nodul rădăcină. În exemplul de mai sus, nodul iniţial are câmpurile:

ActiveNodes[0].color = „alb”

ActiveNodes[0].depth = 3

ActiveNodes[0].code = 000

ActiveNodes[0].parent = ActiveNode[0]

De asemenea se consideră o listă de noduri, corespunzătoare quadtree-ului liniar, notată prin QuadTree, care de asemenea conţine iniţial doar nodul corespunzător blocului format din întreaga imagine.

La fiecare moment dat există un nod activ curent, care se parcurge pixel cu pixel în direcţia orizontală. Iniţial nodul curent – nod_curent – este chiar ActiveNodes[0]. Atrunci când se întâlneşte primul pixel (x,y) de altă culoare decât culoarea nodului curent trebuie creat un nou nod activ, corespunzător blocului care începe la (x,y) şi trebuie eventual spart blocul corespunzător nodului curent (dacă acest lucru nu a fost deja efectuat).

Calcularea codului noului nod activ se realizează în funcţie de (x,y) iar a adâncimii corespunzătoare se realizează astfel:

```
depth = 0, xt=x, yt=y
cat timp xt mod 2 =0 si yt mod 2 = 0 si depth<n
      xt=xt/2
      yt=yt/2
      depth=depth+1
sfarsit cât timp.
```

Dacă depth = 0 atunci blocul care începe la (x,y) conţine un singur pixel, deci nu este nevoie de crearea unui nou nod activ.

Spargerea blocului din care face parte pixelul (x,y) se realizează prin funcţia INSERT, care inserează noile noduri in QuadTree şi elimină nodul corespunzător blocului care a fost spart.

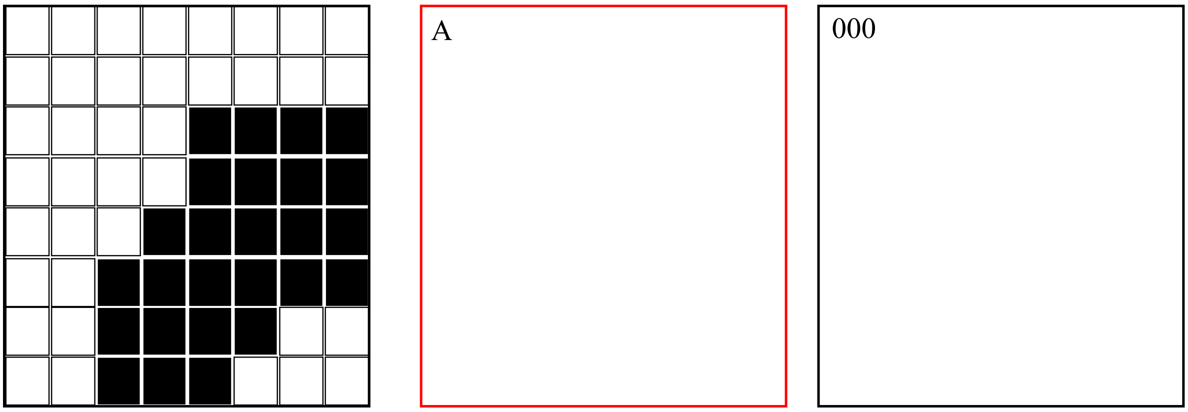
Considerând code = codul calculat din (x,y) atunci

```
INSERT(code)
sparge=true
cat timp(sprge=true)
      k=0
      cât timp (k<QuadTree.dim şi QuadTree[k].code < code)
            k=k+1
      sfarsit cat timp
      k=k-1
      daca QuadTree[k].code = code atunci //blocul deja a fost spart
            sparge = false
      altfel //sparge blocul
            alocă nod2, nod3, nod4
            nod2.code = modif_digit(QuadTree[k].code,
                                    QuadTree[k].depth-1,1)
            //modifica cifra din cod pe poziţia QuadTree[k].depth-1
            //la valoarea 1
            nod2.color = determina_culoarea_primului pixel(nod2.code)
            nod2.depth = QuadTree[k].depth-1.
            nod3.code = modif_digit(QuadTree[k].code,
                                          QuadTree[k].depth-1,2)
            nod4.code = modif_digit(QuadTree[k].code,
                                          QuadTree[k].depth-1,3)
            nod3.color = determina_culoarea_primului pixel(nod3.code)
            nod4.color = determina_culoarea_primului pixel(nod3.code)
            nod3.depth=nod4.depth=nod2.depth
            QuadTree[k].depth= QuadTree[k].depth-1
            Inserează nod2, nod3 şi nod4 după QuadTree[k] în QuadTree
      sfarsit daca
sfarsit cat timp
RETURN
```

Considerând code = codul calculat din (x,y), se determină în QuadTree indicele k, pentru care QuadTree[k].code ≤ code < QuadTree[k+1].code.

Această inegalitate asigură faprul că pixelul (x,y) face parte din blocul QuadTree[k], acest bloc trebuind să fie divizat.

Cât timp QuadTree[k].code< code se înlocuieşte nodul QuadTree[k] cu 4 noduri obţinute prin divizarea blocului reprezentat de către acest nod în 4 blocuri noi şi se recalculează indicele k, astfel încât să satisfacă inegalitatea de mai sus.



ActiveNodes QuadTree

Considerăm exemplul de mai sus:

ActiveNodes = {(000, „alb”, 3, ActiveNodes[0])}

QuadTree = {(000, „alb”, 3)}

nod_curent = ActiveNodes[0] =A (din figură)

Prima coordonată pentru care se determină un pixel negu este (4,2). Codul corespunzător este obţinut prin

- interclasarea codurilor binare ale lui y şi x, deci a codurilor 010 şi 100 => codul binar 011000
- acest cod se transformă în baza 4 şi obţin: 120. Acesta este codul noului nod activ: code=120

Adâncimea noului nod se calculează:

xt = 4, yt = 2, depth =0

xt=2, yt = 1, depth =1

acum yt mod 2≠0 deci ne oprim si adâncimea este depth = 2

Noul nod activ: B = (120, „negru”, 2, ActiveNodes[0])

Lista ActiveNodes = {(000, „alb”, 3, ActiveNodes[0]), (120, „negru”, 2, ActiveNodes[0])}

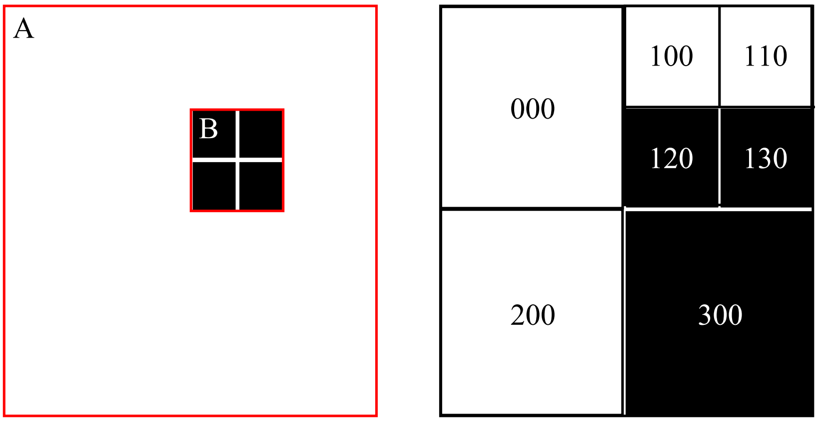
Apelăm INSERT(120)

- se determină k = 0
- QuadTree[0].code = 000 ≠ 120 deci se sparge blocul în 4 blocuri cu codurile 000, 100, 200, 300 şi se inserează în QuadTree conform algoritmului
- Deci QuadTree = {(000, „alb”, 2), (100, „alb”, 2), (200, „alb”, 2), (300, „negru”, 2)}

- Se calculează un nou k = 1, QuadTree[1].code = 100 ≠ 120, deci blocul cu codul 100 se sparge din nou în patru blocuri şi se obţine

  QuadTree = {(000, „alb”, 2), (100, „alb”, 1), (110, „alb”, 1) , (120, „negru”, 1), (130, „negru”, 1) ,(200, „alb”, 2), (300, „negru”, 2)}

- Noul k=3, iar QuadTree[3].code = 120, deci sparge=false şi se încheie operaţia de inserţie



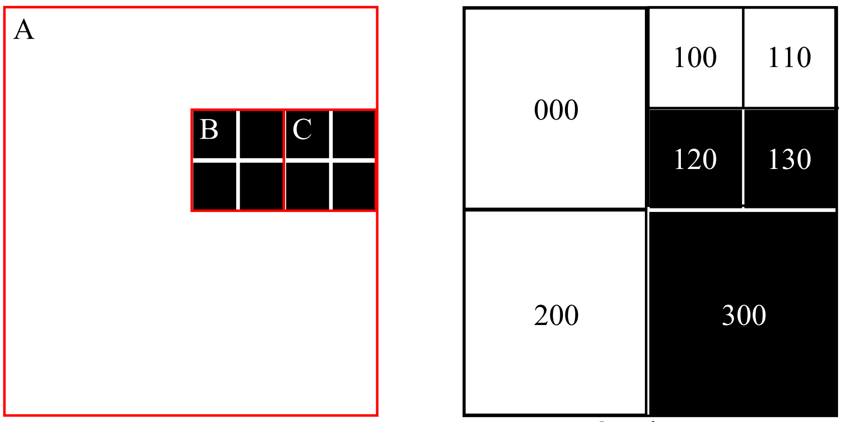
ActiveNodes QuadTree

Nodul activ corespunzător pixelului (5,2) este ActiveNodes[1] şi are aceeaşi culoare ca şi nodul activ, deci se trece la pixelul următor.

Nodul activ corespunzător pixelului (6,2) este ActiveNodes[0]. Culoarea lui este „negru” ≠ ActiveNodes[0].color => se crează un nou nod activ C = (130, „negru”,1, ActiveNodes[0]), deci:

ActiveNodes = {(000, „alb”, 3, ActiveNodes[0]), (120, „negru”, 2, ActiveNodes[0]), (130, „negru”,1, ActiveNodes[0])}

Se apelează INSERT(130) şi se va constata că deja există nodul respectiv în QuadTree, deci nu mai e necesară spargerea.



ActiveNodes QuadTree

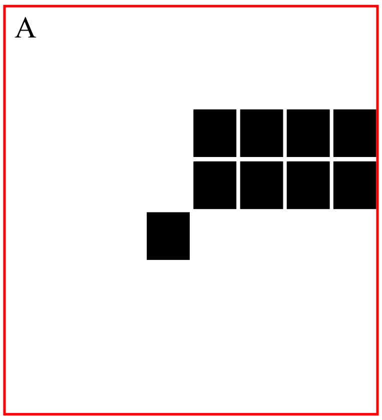
Pixelul (7,2) are aceeaşi culoare cu cea a nodului activ corespunzător, la fel şi pixelii (0,3), (1,3), (2,3), (3,3), (4,3), (5,3) (6,3) şi (7,3).

În plus, când se ajunge la pixelul (5,3) se observă că s-a încheiat blocul activ cu codul 120, deci acesta se elimină din lista blocurilor active. La fel, la pixelul (7,3) se incheie blocul activ cu codul 130, deci şi acesta se elimină din lista de blocuri active.

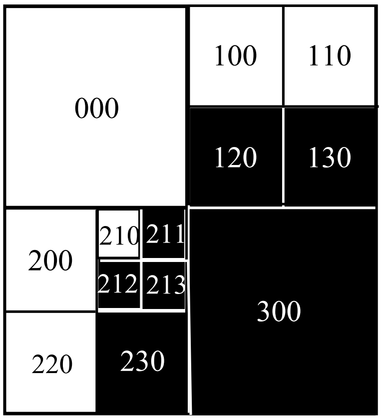
Se obţine în acest moment lista de noduri active:

ActiveNodes = {(000, „alb”, 3, ActiveNodes[0])}

În continuare primul pixel de culoare diferită de cea a blocului activ este (3, 4). Ar urma să fie creat un nou nod activ: 100 cu 011 => 100101 >= 211 => (211, „negru”,0, ActiveNodes[0]), dar cum depth=0, nodul conţine un singur pixel, deci nu e necesară inserţia în lista de noduri active, deoarece s-a şi închieat parcurgerea lui.



ActiveNodes



QuadTree

În schimb se apelează INSERT(210)

QuadTree={(000, „alb”, 2), (100, „alb”, 1), (110, „alb”, 1) , (120, „negru”, 1), (130, „negru”, 1) ,(200, „alb”, 2), (300, „negru”, 2)}

- k =5, QuadTree[5].code = 200 ≠211, deci se sparge blocul corespunzător în 4 blocuri şi se obţine

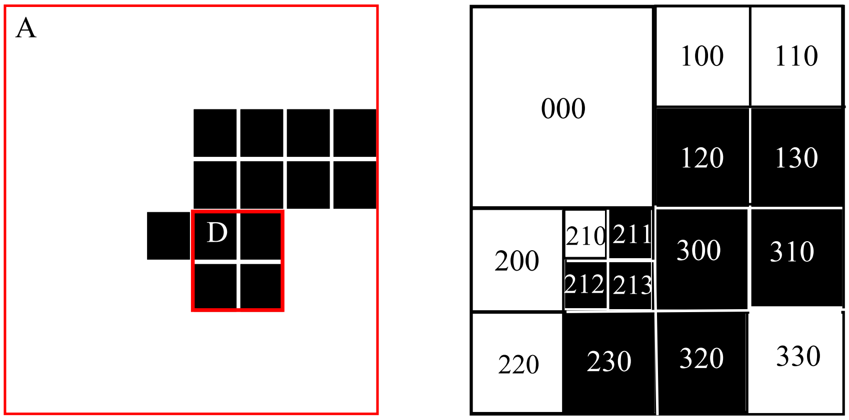
  QuadTree={(000, „alb”, 2), (100, „alb”, 1), (110, „alb”, 1) , (120, „negru”, 1), (130, „negru”, 1) ,(200, „alb”, 1), (210, „alb”, 1), (220, „alb”, 1), (230, „negru”, 1) ,(300, „negru”, 2)}

- Noul k = 6, QuadTree[6].code = 210 ≠211, deci se sparge blocul cu codul 210 în 4 blocuri şi se obţine:

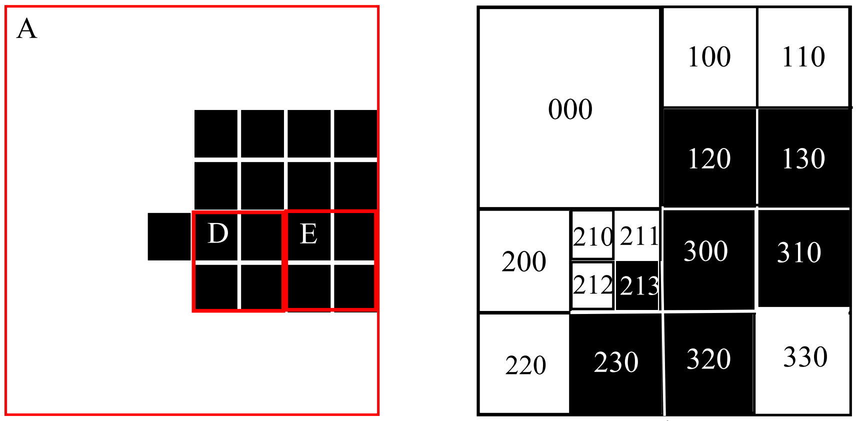
- QuadTree={(000, „alb”, 2), (100, „alb”, 1), (110, „alb”, 1) , (120, „negru”, 1), (130, „negru”, 1) ,(200, „alb”, 1), (210, „alb”, 0), (211, „negru”, 0), (212, „negru”, 0), (213, „negru”, 0) , (220, „alb”, 1), (230, „negru”, 1) ,(300, „negru”, 2)}

- În acest moment k = 7 şi QuadTree[7].code = 211 deci se opreşte procedura

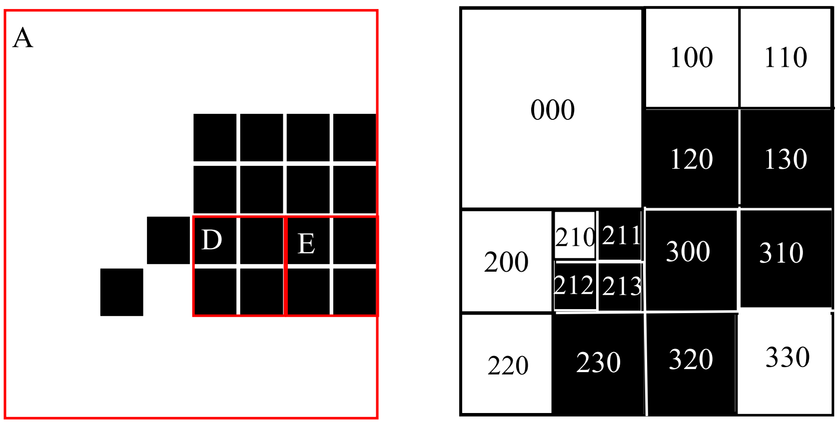
Continuând algoritmul descris mai sus:



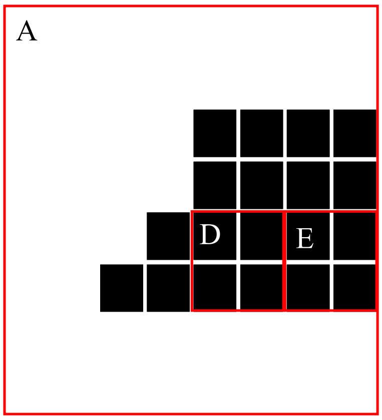
ActiveNodes QuadTree



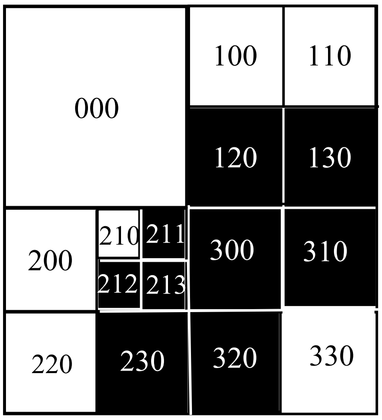
ActiveNodes QuadTree



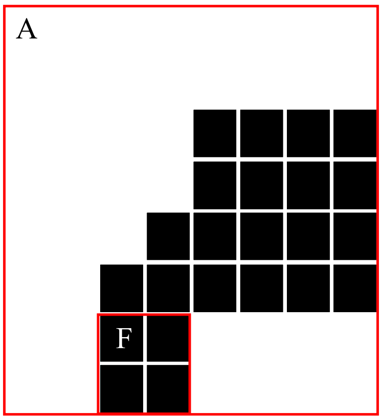
ActiveNodes QuadTree



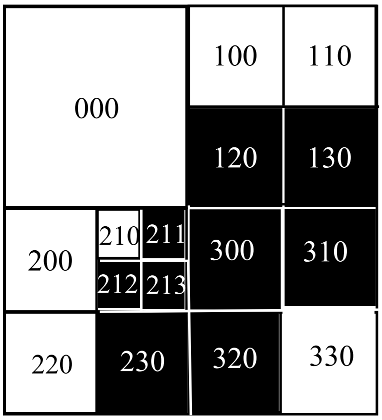
ActiveNodes



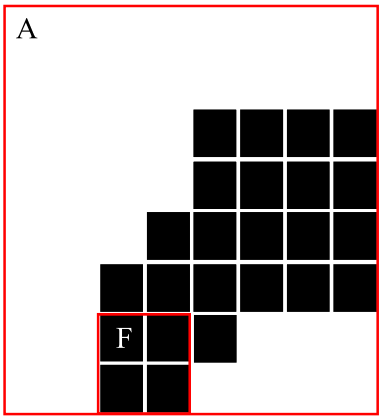
QuadTree



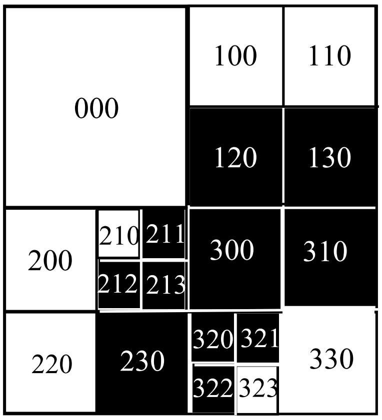
ActiveNodes



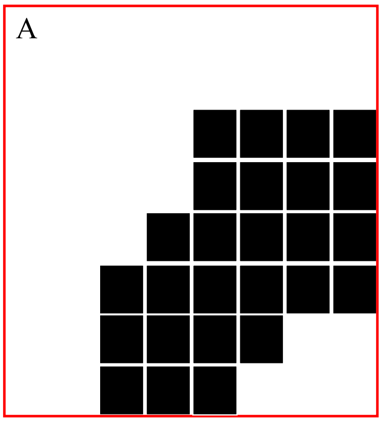
QuadTree



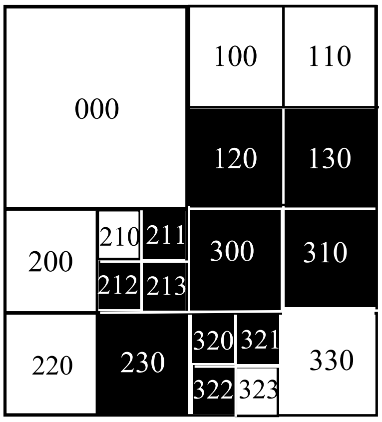
ActiveNodes



QuadTree



ActiveNodes



QuadTree

se ajunge în final la:

- QuadTree={(000, „alb”, 2), (100, „alb”, 1), (110, „alb”, 1) , (120, „negru”, 1), (130, „negru”, 1) ,(200, „alb”, 1), (210, „alb”, 0), (211, „negru”, 0), (212, „negru”, 0), (213, „negru”, 0) , (220, „alb”, 1), (230, „negru”, 1), (300, „negru”, 1), (310, „negru”, 1), (320, „negru”, 0), (321, „alb”, 0), (322, „alb”, 0), (323, „alb”, 0), (330, „alb”, 1)}.

Algoritm

```
CREATE_LINEAR_QUADTREE(Image, n)
// QuadTree = lista de frunze returnate
//LIST = vector de pointeri, in care pentru fiecare coloana j,
//LIST[j/2] = nodul activ corespunzator pixelului de pe coloana j
//din randul curent

color = Image[0][0] //culoarea primului pixel din imagine
code = 00...00 (0 de n ori)
ActiveNodes[0] = new nod(code, color, n, ActiveNodes[0])
pentru j=0, 2^(n-1) -1
      LIST[j] = ActiveNodes[0]
sfarsit pentru
pentru row=0, 2^n -1
      pentru col = 0, 2^n -1
            nod_curent = LIST[col/2]
            daca nod_curent.color ≠ Image[row][col] atunci
                  xt = col, yt = row, depth = 0
                  cat timp xt mod 2=0 si yt mod 2 =0 si depth < n
                        xt=xt/2
                        yt=yt/2
                        depth=depth+1
                  sfarsit cat timp
                  daca depth ≠ 0 atunci
                        nod_activ_nou = new nod
                        nod_activ_nou.color = Image[row][col]
                        nod_activ_nou.depth = depth
```

```
                    nod_activ_nou.parent = nod_curent
                    pentru j=col/2, 2^(depth-1) -1
                         LIST[j]= nod_activ_nou
                    sfarsit pentru
                    add(nod_activ_nou, ActiveNodes)
               sfarsit pentru
               code = calculate_code(row,col)
               INSERT(code)
          sfarsit daca
          cat timp (row+1) mod (2^(nod_curent.depth)) = 0 si (col+1) mod
                                                   (2^(nod_curent.depth)) = 0
          //s-a inchieat un nod activ
               T = nod_curent.depth
               pop(nod_curent, ActiveNodes)
               nod_curent=nod_curent.parent
          sfarsit cat timp
          pentru j=col/2, col/2 – 2^(T-1)+1, pas -1
               LIST[j] = nod_curent
          sfarsit pentru
     sfarsit pentru
sfarsit pentru
RETURN
```

///GATA

Determinarea vecinilor la QuadTree liniar – calculul codului + căutarea în lista de frunze.

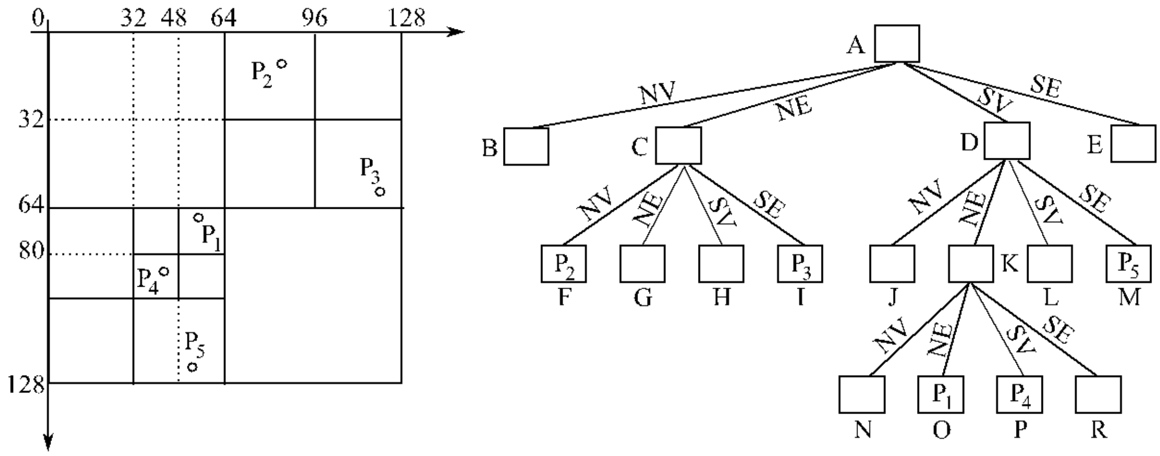
Point Region Quad Trees - PR-Quadtrees

Arborii quad de tip point-region împart o suprafaţă pătrată în mod similar ca şi arborii quad pentru suprafeţe/imagini, cu deosebirea că, împărţirea pe baza culorii suprefeţei, ci pe baza unui set de puncte plasate în suprafaţa dată.

Ideea generală: se consideră o suprafaţă bidiemnsională cu coordonatele (x,y) în domeniul [0,M]× [0,M] şi n puncte $P_i=(x_i,y_i)$, i=1, n plasate pe această suprafaţă.

Suprafaţa se împarte în patru suprafeţe egale în mod recursiv, până când fiecare frunză conţine cel mult unul dintre punctele P_i.

Exemplu:



Căutarea unui nod:

```
PR_SEARCH(T,P) //caut nodul in care s-ar plasa punctul P
daca P.x<0 sau P.x>T->bottom_right.x sau P.y<0 sau
                      P.y>T->bottom_right.y atunci
RETURN NULL
Z=T
cat timp Z nu e frunza
      daca P.x ≤ Z->bottom_right.x/2 si P.y ≤ Z->bottom_right/2
      atunci
                  Z=Z->NV
            altfel
                  daca P.x>Z->bottom_right.x/2 si
                                 P.y > Z->bottom_right/2 atunci
                              Z=Z->SE
                  altfel
                        daca P.x ≤ Z->bottom_right.x/2 atunci
                              Z=Z->SV
                        altfel Z=Z->NE
```

```
                        sfarsit daca
                    sfarsit daca
                sfarsit daca
sfarsit cat timp
RETURN Z
```

Exemplu: considerăm arborele din imagine, construit pe baza setului de puncte {(55,67),(80,15),(119,58),(41,85),(53,117)}

SEARCH(T, (50,105))

Z=A nu e frunză => verific în care dintre cele 4 blocuri descendent poate fi găsit P=(50,105)

$$\left.\begin{array}{l} 0 < P.x = 50 < 64 \\ 64 < P.y = 105 < 128 \end{array}\right\} \Rightarrow Z = Z \rightarrow SV = D$$

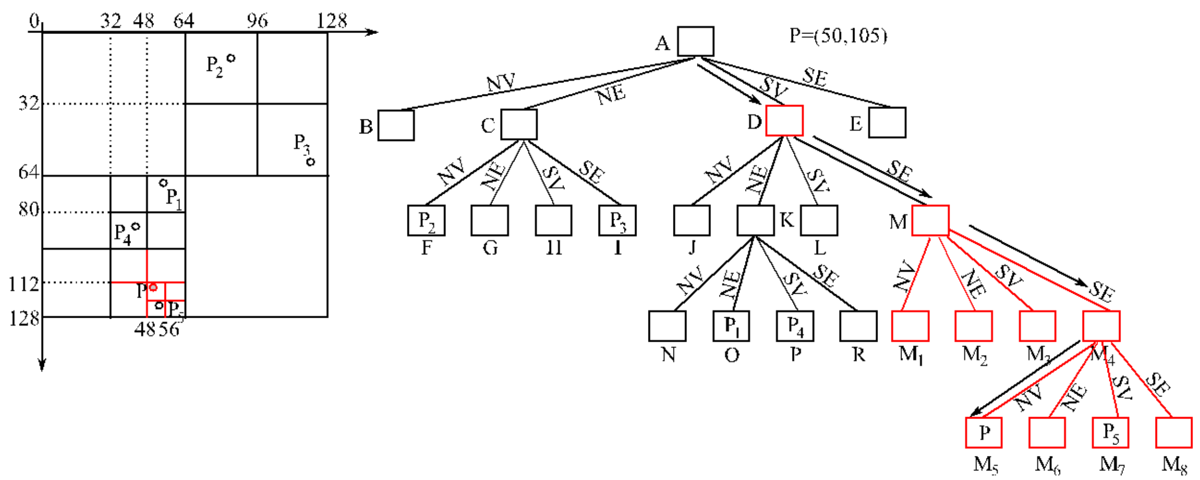
D nu este frunză => verific în care dintre cele 4 blocuri descendente se plasează P

$$\left.\begin{array}{l} 32 < P.x = 50 < 64 \\ 96 < P.y = 105 < 128 \end{array}\right\} \Rightarrow Z = Z \rightarrow SE = M$$

M este frunza corespunzătoare puctului (50,105). Punctul P_5 aflat în M este cel mai apropiat puncte de P.

Dacă se doreşte inserarea punctului P într-un PR-arbore quad, atunci, după identificarea frunzei M care corespunde pnctului P, se verifică dacă frunza respectivă conţine deja un punct, în exemplul anterior, P_5:

- Daca nu, atunci se inserează P în câmpul informaţie
- Altfel (cazul exemplului) se sparge blocul corespunzător frunzei M în 4 blocuri noi, frunza se înlocuieşte cu un nod interior, care are 4 descendenţi, corespunzători noilor blocuri obţinute, se plasează punctul aflat in frunza M în nodul nou construit corespunzător şi se continuă procesul de căutare a unei frunze corespunzătoare lui P, cu evetuale noi spargeri, dacă este necesar.



```
PR_INSERT(T, point, M)
daca point.x<0 sau point.x>M-1 sau point.y<0 sau point.y>M atunci
      Scrie(„nu poate fi inserat”)
      RETURN
sfarsit daca
daca T = NULL atunci
      Aloca T
      T->info =point
      T->top_left = (0,0)
      T->bottom_right = (M-1,M-1)
      T->SV=T->SE=T->NV=T->NE=NULL
      RETURN
sfarsit daca
X=T
repeta
      //cauta frunza in care se plaseaza punctul point
      Z=PR_SEARCH(X,point)
      daca Z->info = NULL atunci
            Z->info=point
      altfel
         • sparge suprafata din Z in patru suprafete
           plasate in noduri noi Z1, Z2, Z3 si Z4
         • plaseaza Z->info in nodul corespunzator Z1,Z2,Z3,Z4

         • Z->NV=Z1, Z->NE=Z2, Z->SV=Z3, Z->SE=Z4
         • Z->info = NULL
      sfarsit daca
      X=Z
      //acum Z nu mai este frunza si se reia cautarea
pana cand X = frunza
RETURN
```

Construcţia unui Arbore quad PR se realizează prin inserţii succesive într-un arbore iniţial vid.

```
PR-CONSTRUCT(P,n,M)
Initializare T
pentru i=1, n
      PR_INSERT(T, P[i])
sfarsit pentru
RETURN T
```

Eliminarea unui punct P:

Se caută frunza M corespunzătoare nodului P în arbore. Dacă M nu conţine P, atunci P nu există în arbore şi deci nu poate fi şters.

Altfel: se şterge P din frunza M.

Dacă fraţii lui M sunt frunze şi în blocul reprezentat prin aceste 4 frunze se află cel mult un punct Q din mulţimea de puncte ale PR-arborelui, atunci cele patru frunze se şterg, iar punctul Q se mută în părintele lui M, care acum devine frunză.

Această etapă se repetă, până când se ajunge la un nod ale cărui descendenţi nu sunt cu toţii frunze sau sunt 4 frunze care conţin împreună cel puţin 2 puncte.

Utilizăm funcţia IS_MERGEABLE(Y) care verifică dacă nodul Y conţine 4 descendenţi care pot fi eliminaţi şi în caz afirmativ returnează informaţia (dacă există) a unicului descendent frunză care nu este vid.

```
IS_MERGEABLE(Y)
      nr =0
      info=NULL
      daca Y->NV≠ frunza sau Y->NE ≠ frunza sau Y->SV≠ frunza sau
                              Y->SE≠ frunza atunci
            RETURN NULL
      daca Y->NV->info≠ NULL atunci
            info=Y->NV->info
            nr=nr+1
      daca Y->NE->info≠ NULL atunci
            info=Y->NE->info
            nr=nr+1
      daca Y->SV->info≠ NULL atunci
            info=Y->SV->info
            nr=nr+1
      daca Y->SE->info≠ NULL atunci
            info=Y->SV->info
            nr=nr+1
      daca nr>1 atunci info=NULL
RETURN info
```

```
PR_DELETE(T,P)
Z=PR_SEARCH(T,P)
daca Z->info = P atunci
      Z->info = NULL
      Y=Z->p
      ok=true
      cat timp Y≠NULL  si ok
            Info= IS_MERGEABLE(Y)
            daca Info = NULL atunci
                  ok=false
            altfel
                  Y->info = Info
                  Y->NV= Y->NE= Y->SV= Y->SE =NULL
                  Z=Y
```

```
                Y=Y->p
            sfarsit daca
        sfarsit cat timp
altfel
        scrie(„nu exista P”)
sfarsit daca
RETURN
```

Exemplu ştergem punctul (50,105) din arborele care conţine punctele: {P1=(55,67), P2=(80,15), P3=(119,58), P4=(41,85), P5=(53,117), P6=(50,105)}

Observaţie:

1. forma unui PR-QuadTree nu depinde de ordinea în care sunt inserate punctele în arbore.
2. Dacă se inserează două puncte foarte apropiate este posibil să trebuiască efectuate numeroase spargeri ale blocurilor, până se ajunge la separarea celor două puncte, ceea ce conduce la creşterea în adâncime a arborelui şi la introducerea unui număr semnificativ de noduri ce nu conţin nici un punct.

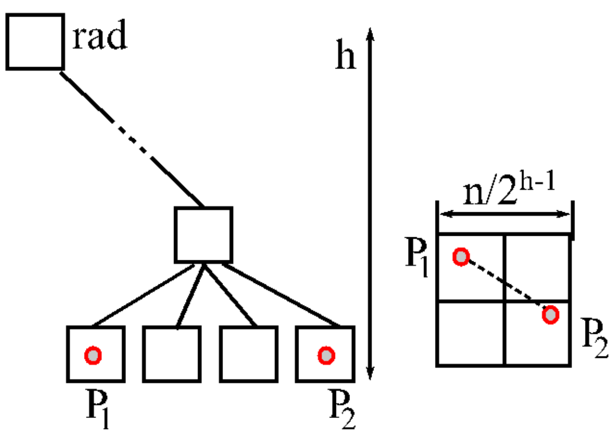
Înălţimea unui PR-QuadTree:

Considerând d=distanţa minimă între două puncte din blocul original şi n dimensiunea laturii acestui bloc, atunci adâncimea h a PR-Quadtree-ului corespunzător este:

$$\log_4 n \le h \le \log_2\left(\sqrt{2}n/d\right)+1$$

Demonstraţie: evident, dacă toate frunzele sunt la aceeaşi adâncime, h=$\log_4$n.

Altfel: cele mai mici blocuri, adică cele mai adânci frunze se obţin pentru separarea celor mai apropiate două puncte.



Presupunând că aceste două puncte P1 şi P2 se află la adâncimea h, părintele din care au provenit se afla la dâncimea h-1 şi avea dimensiunea laturii $L=n/2^{h-1}$. Distanţa între P1 şi P2 este d şi

$$d \le \sqrt{2}L = \sqrt{2}n / 2^{h-1}$$

De aici rezultă

$$h - 1 \le \log_2\left(\sqrt{2}n / d\right) \Rightarrow h \le \log_2\left(\sqrt{2}n / d\right) + 1$$

Determinarea vecinilor: acelaşi algoritm ca şi pentru Quadtrees.

Exemplu de aplicaţie:

Considerând un bloc pătrat de dimensiune 2n×2n ce reprezintă o hartă, pe care se află amplasate m oraşe. Acestea pot fi reprezentate cu ajutorul unui PR-arbore quad. Să se determine toate oraşele aflate la distanţa maximă d de un oraş dat P.

Rezolvare:

Cobor în arbore până la cel mai mic nod intern M->top_left.x ≤ (P.x-d)sqtr(2),

M->top_left.y ≤ (P.y-d)sqtr(2), M->bottom_right.x ≥ (P.x+d)sqtr(2), M->bottom_right.y ≥ (P.y+d)sqtr(2).

Se testează toate oraşele aflate în subarborele de rădăcină M.

Variantă: Bucket-QuadTree – pentru a evita situaţia creşterii semnificative a înălţimii arborelui şi generării de multe frunze vide în cazul inserţiei de puncte foarte apropiate, poate fi folosită o variantă de Quadtree care să permită stocarea a un număr b>1 de puncte într-un nod. Acest lucru conduce la spargerea unui bloc, daor dacă el conţine mai mult de b puncte.

Arbori Kd - Kd-Trees

= arbori k-dimensionali

Arborii kd au fost elaboraţi pentru stocarea şi manipularea eficientă de date k-dimensionale. Iniţial k se limita la 2 dimensiuni = puncte în plan sau 3 diemsiuni = puncte în spaţiu, dar a fost realizată extensia la oricâte dimensiuni.

Arborii kd sunt arbori binari în care fiecare nod împarte spaţiu k-dimensional printr- un hiperplan perpemdicular pe direcţia uneia dintre axele de coordonate. Punctele aflate de-o parte a hiperplanului vor fi plasate în descendentul stâng iar celelate în descendentul drept.

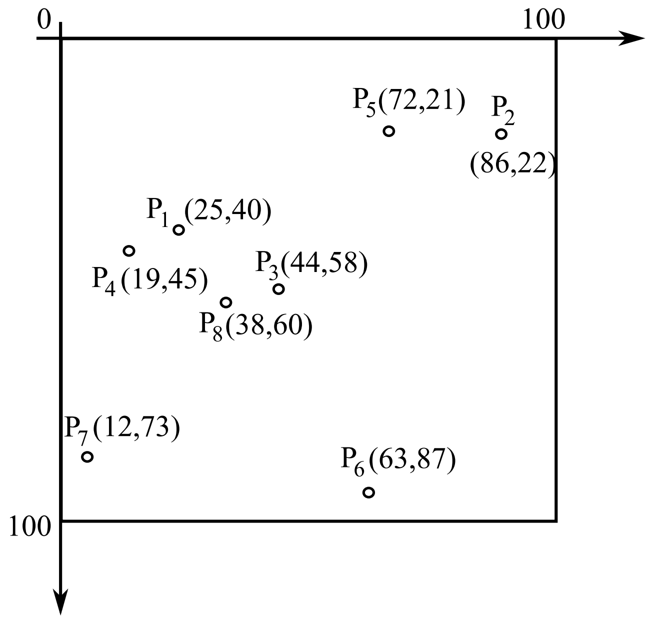
Direcţia după care se selectează hiperplanul în fiecare nod este aleasă de obicei depinzând de adâncimea nodului în arbore. Îniţial separarea spaţiului se realizează după direcţia dată de prima dimensiune a punctelor, pe nivelul următor de următoarea dimensiune etc., după care se reia ciclic începând de la prima dimensiune.

Exemplu: dacă k=3 avem 3 axe: 0x,0y,0z. Fiecare punct are trei coordonate (x,y,z). Iniţial se face separarea după direcţia axei 0x, la următorul nivel după 0y, la următorul nivel după 0z, după care se reia cu separarea după direcţia 0x...

Deci

- dacă se consideră un punct P=(x0,x1,...,xk-1), atunci direcţia după care se va face separarea în subspaţii într-un nod Z de adâncime j este dir(Z) = axa j mod k. Toate punctele cu componenta a j-a < xj vor fi plasate în subarborele stâng, iar toate cu componenta a j-a >xj vor fi plasate în subarborele drept.

- În toate nodurile de adâncime j, discriminarea se face după aceeaşi direcţie



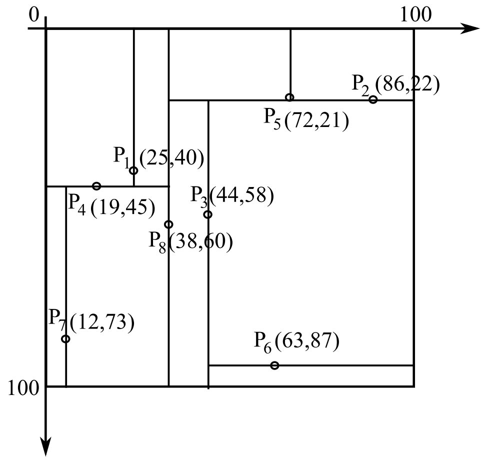
Se consideră mulţimea de puncte A din figură,
A={(25,40),(86,22),(44,58),(19,45),(72,21),(63,87),(12,73),(38,60)}

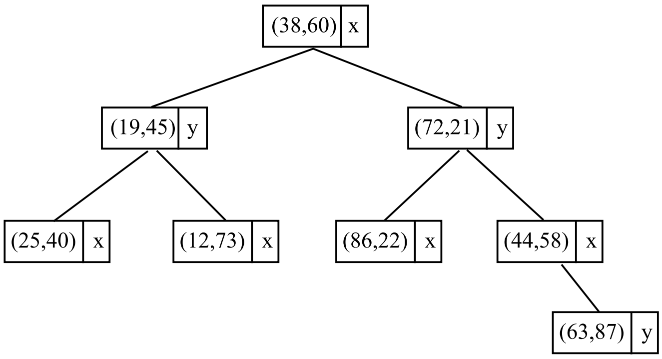
Un arbore kd, k=2 pentru A poate fi contruit după cum urmează.

În rădăcină se plasează punctul (38,60) – discriminarea se face după 0x, punctele care au coordonata x mai mică decât 38, {(25,40),(19,45),(12,73)}, se plasează în subarborele stâng, cele cu coordonata x mai mare decât 38, {(86,22),(44,58),(72,21),(63,87)} se plasează în subarborele drept.

Procedeul se repetă recursiv pentru {(25,40),(19,45),(12,73)} şi {(86,22),(44,58),(72,21),(63,87)}

Se obţine următoarea împărţire a spaţiului şi arborele corespunzător:





Observaţie: cum s-a făcut alegerea primului punct de inserţie: s-a ales mereu valoarea mediană faţă de direcţia curentă. Acest lucru nu este obligatoriu, dar consuce la un arbore echilibrat.

Problemă: considerând nodul curent Z ce conţine punctul P=(p0,p1,...,pk-1), dir(Z) = j şi un punct Q=(q0,q1,...,qk-1) în care subarbore al lui Z se va plasa Q dacă pj=qj? Există mai multe soluţii propuse, de exemplu:

- definirea unei superchei, care face discriminarea între Q şi P şi selectează subarborele pe care se continuă cererea în funcţie de celelalte componente ale vectorilor k-dimensionali
- inserarea pe stânga a elementelor „mai mici sau egale” şi pe dreapta a celor „mai mari”.

Construcţia recursivă a unui kd-tree echilibrat dintr-o mulţime de puncte k-dimensionale.

```
BULD_KD_TREE(A, j)    //A=mulţimea de noduri, j = adancimea radacinii
                      //subarborelui curent
daca A =∅ atunci
      RETURN NULL
 altfel
      P = mediana(A,j) //val mediana fata de componenta a j-a a
vectorilor k-diemansionali
```

```
      A1 = stanga(A,P,j) //returneaza punctele din A care se plaseaza
in subarb st
      A2 = dreapta(A,P,j) //returneaza punctele din A care se
plaseaza in subarb dr
      Z->info = P
      Z->directie=j
      Z->st=BUILD_KD_TREE(A1, (j+1) mod k)
      Z->dr=BUILD_KD_TREE(A2, (j+1) mod k)
sfarsit daca
RETURN Z
```

Funcţia stanga(A,P,j) returnează punctele din a care trebuie inserate pe stânga lui P, iar funcţia dreapta(A,P,j) returnează punctele care trebuie inserate pe dreapta relativ la direcţia j. Aceste funcţii trebuie să ia în considerare şi situaţia valorilor corespunzătoare direcţiei j egale.

Inserţia

```
KD_INSERT(T,P)
daca T.rad=NULL atunci
      Aloca T.rad
      T.rad->info = P
      T.rad->directie=0
      T.rad->st=T.rad->dr=NULL
altfel
      X=T.rad
      cat timp X!=NULL
            Y=X
            daca X->info = P atunci RETURN sfarsit daca
            //functia descendent returneaza pe care parte a unui nod
            //se insereaza un anumit punct
            daca descendent(X,P) = stanga atunci
                  X=X->st
            altfel X=X->dr
            sfarsit daca
      sfarsit cat timp
      aloca Z
      Z->info=P
      Z->directie=(Y->directie+1) mod k
      Z->st=Z->dr=NULL
      daca descendent(Y,P) = stanga atunci
            Y->st=Z
      altfel Y->dr=Z
      sfarsit daca
sfarsit daca
RETURN
```

Aplicaţii - căutarea rapidă de puncte sau regiuni chiar cu specificaţii parţiale:

- valori precizate doar pentru unele dintre câmpuri
- valori ce satisfac inegalităţi (ex: toate punctele aflate la o anumită distanţă de un punct dat)

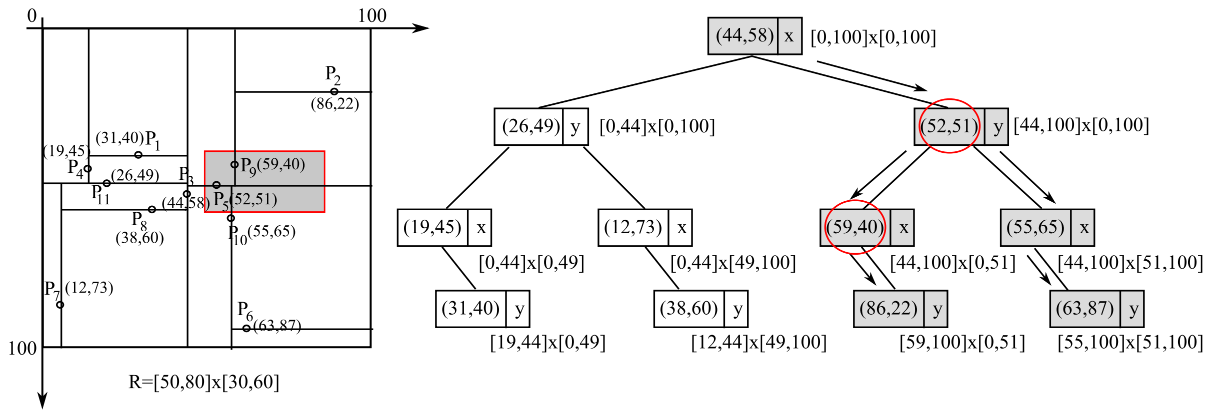
Căutarea unui punct

Căutarea tutror punctelor aflate într-o regiune R

```
KD_SEARCH_REGION(Z,R) //cauta toate punctele din subarborele de rad.
Z incluse în R
verifica daca Z->info este in R
daca Z->st ≠ NULL atunci
      //region(X) reprezinta regiunea corespunzatoare subarborelui
      //de radacina X
      //aceasta poate fi retinuta in nod sau poate fi calculata
      //in timpul parcurgerii de la radacina catre X
      daca region(Z->st) ⊂ R atunci
            afisaza toate punctele din subarborele Z->st
      altfel
            daca region(Z->st) ∩ R≠∅ atunci
                  KD_SEARCH_REGION(Z->st,R)
            sfarsit daca
      sfarsit daca
sfarsit daca
daca Z->dr ≠ NULL atunci
      daca region(Z->dr) ⊂ R atunci
            afisaza toate punctele din subarborele Z->dr
      altfel
            daca region(Z->dr) ∩ R≠∅ atunci
                  KD_SEARCH_REGION(Z->dr,R)
            sfarsit daca
      sfarsit daca
sfarsit daca
RETURN
```

Exemplu: Căutarea tuturor punctelor bidimensionale aflate în regiunea R=[50,80]×[30,60], în arborele din figură, care conţine mulţimea de puncte:

A={(31,40),(86,22),(44,58),(19,45),(52,51),(63,87),(12,73),(38,60),(59,40),(55,65),(26,49)}



Pornim de la Z=radacina, result=∅, R=[50,80]×[30,60], Z->info=(44,58) ∉ R

- verific region(Z->st) = [0,44]×[0,100] care nu se intersectează cu R

-verific region(Z->dr) = [44,100]×[49,100] care se intersectează cu R => KD_SEARCH_REGION(Z->dr,R) => acum Z->info=(52,51) ∈ R => result = {(52,51)}

- verific region(Z->st) = [44,100]×[0,51] care intersectează R => merg la Z->st => acum Z->info = (59,40) ∈R => result = {(52,51),(59,40)}

- Z->st = NULL

- verific region(Z->dr) = [59,100]×[0,51] care intersectează R => merg la

Z->dr, Z->info = (86,22) ∉ R

- verific region(Z->dr) = [44,100]×[51, 100] care intersectează R =>merg la Z->dr => acum Z->info = (55,65) ∉ R

- Z->st=NULL

- verific region(Z->dr) = [55,100]×[51, 100] care intersectează R => merg la Z->dr => acum Z->info=(63,87)∉R

Algoritmul s-a incheiat cu result = {(52,51),(59,40)}

Căutare cu potrivire parţială

Enunţ: să se determine toate punctele P=(p_0,p_2,...,p_{k-1}) pentru care p_{ij}=a_{ij}, j=0,t, 0 < t < k-1, ij∈{0,1,…,k-1}, a_{ij} = valori date.

Algoritm:

```
KD_SEARCH_PARTIAL(Z, {i0,i1,..,it},{ai0,ai2,…,ait}, result)
daca Z->info(ij) = aij pentru ∀j∈{0,1,…,t} atunci
     Add(rezult,Z)
sfarsit daca
```

```
daca Z->directie = ij ∈ {i0,i1,..,it} atunci
      daca aij < Z->info(ij) atunci
            KD_SEARCH_PARTIAL(Z->st,{i0,i1,..,it},{ai0,ai2,…,ait},
                                                             result)
      altfel
            daca aij > Z->info(ij) atunci
              KD_SEARCH_PARTIAL(Z->dr,{i0,i1,..,it},{ai0,ai2,…,ait},
                                                             result)
            altfel
              KD_SEARCH_PARTIAL(Z->st,{i0,i1,..,it},{ai0,ai2,…,ait},
                                                             result)
              KD_SEARCH_PARTIAL(Z->dr,{i0,i1,..,it},{ai0,ai2,…,ait},
                                                             result)
            sfarsit daca
      sfarsit daca
sfarsit daca
RETURN
```

Exemplu:

Observaţii:

Există şi alte tipuri de operaţii pe arbori kd, de exemplu căutarea minimului după o anumită direcţie, căutarea celui mai apropiat punct de un punct dat sau ştergerea unui punct, operaţii pe care însă nu le vom discuta în cadrul cursului.