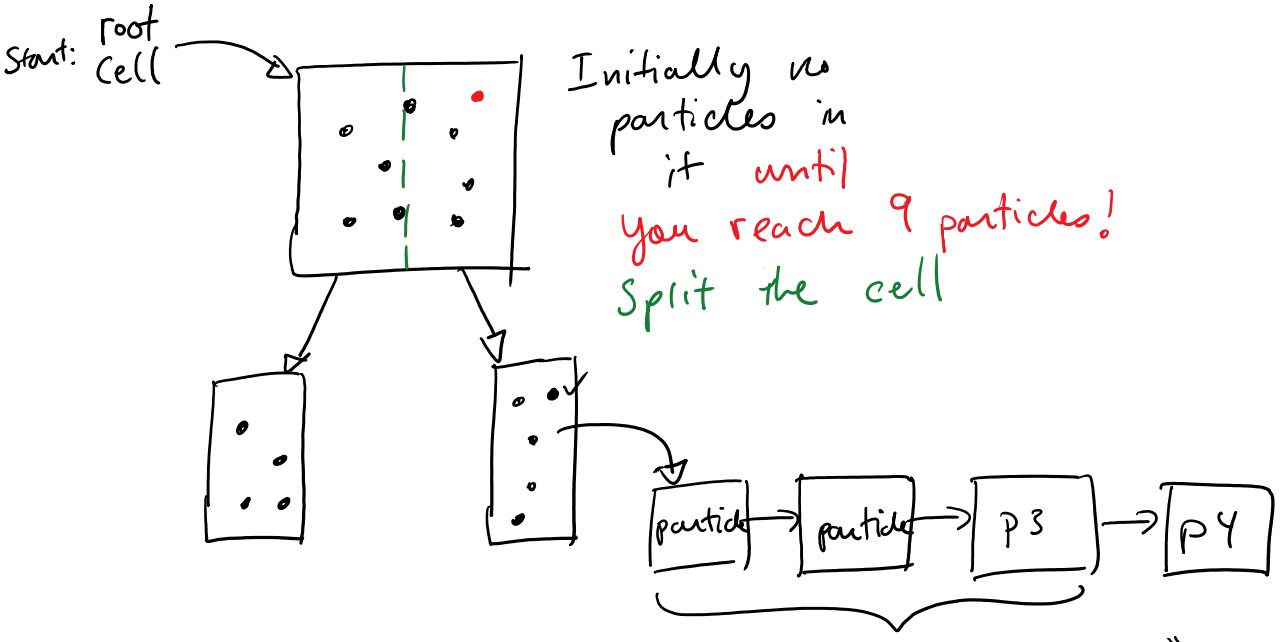
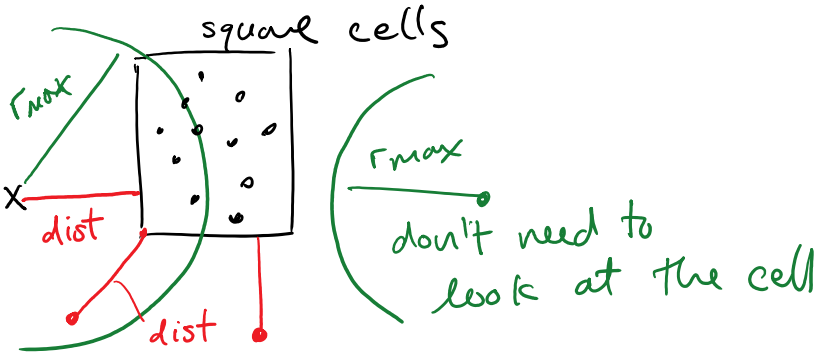
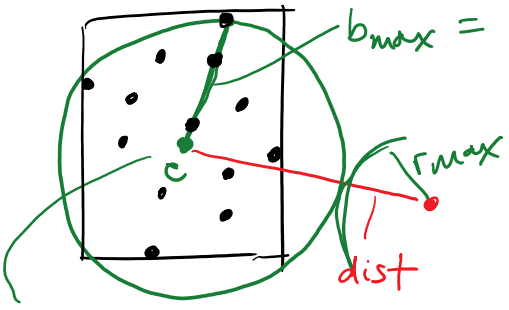
Another tree construction method:



Partition function

↳ insures that spatial locality = memory locality.

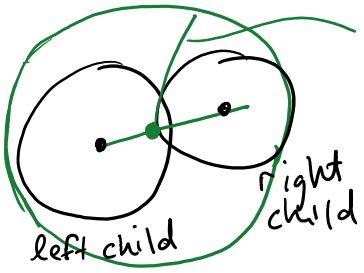




then ignore the cell.

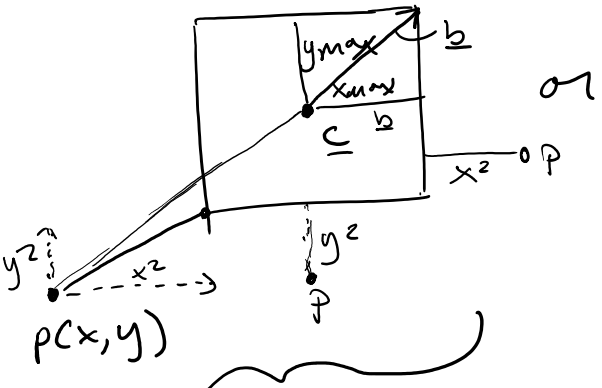
★ The best center is the one of the tightest bounding ball.

★ Only need to do this for leaf cells.

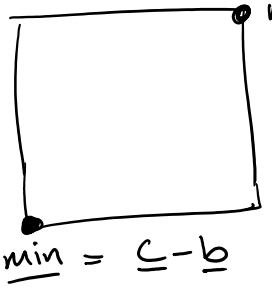


parent cell's bmax (easy to calculate)

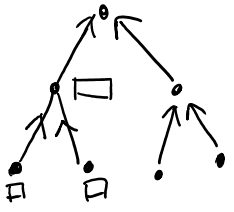
Ball-box Test again:



or

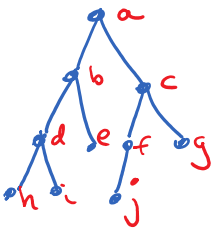



```
d2 = 0
for (d = 0; d < 2; ++d) {
    t = abs(c[d] - p[d]) - b[d]
    if (t > 0)  d2 += t*t;
}
return d2
```



Recursion was made for tree algorithms!

LNR - left-node-right traversal of tree!



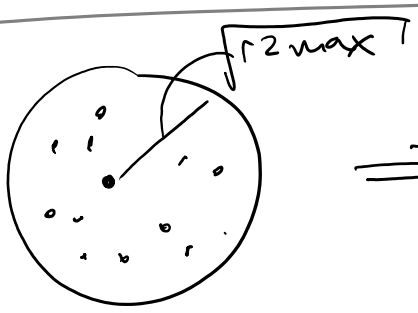

```
lnr(c)
    if ( c!=NULL) {
        lnr(c->left)
        print(c->data)
        lnr(c->right)
```

⇒ hdibeajfcg

NLR ⇒ abdhiecfjg

"WALKING the Tree"

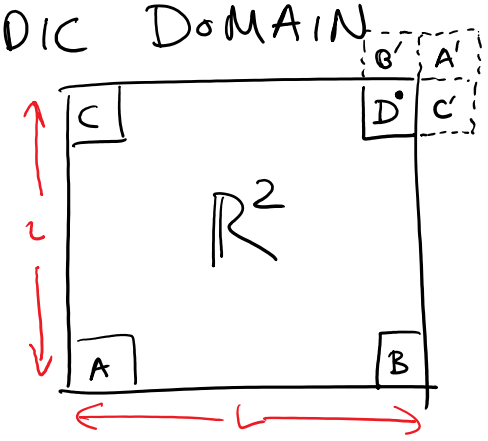
```
def BALLWALK(c, r, r2max):
    cnt = 0                                  # r2max: square of ball radius
    if c.isLeaf:
        for a in A[c.lower : c.upper+1]:
            if dist2(r, a.r) < r2max:        # careful for method2
                ++cnt
    else:
        if c.left.dist2(r) < r2max:          # stays the same for method 2
            cnt += BALLWALK(c.left, r, r2max)
        if c.right.dist2(r) < r2max:
            cnt += BALLWALK(c.right, r, r2max)
    return cnt
```

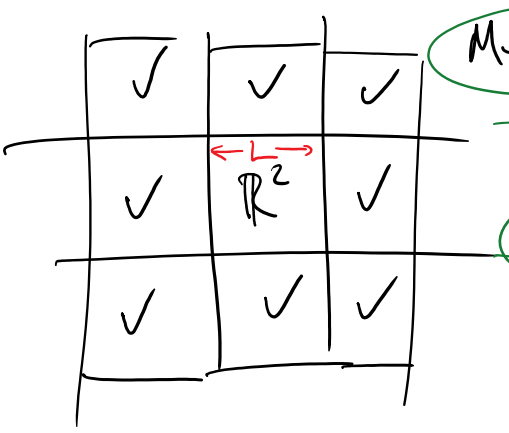


⇒ returns the number of particles within distance rmax.

of particles ...
distance rmax.

PERIODIC DOMAIN



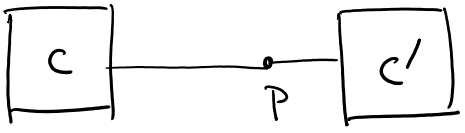


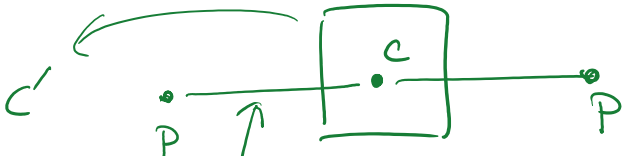
Method 1

or

Method 2 → change the c.dist2 function.

works as long as the rmax is < L/2 !





c-p is positive ⇒ c-L
c-p is negative ⇒ c+L

```
d2 = 0
for (d=0; d<2; ++d) {
    t = c[d] - p[d];
    if (t<0)  t1 = t + L
```

```
else  t1 = t - L
t = abs(t) - b[d]
t1 = abs(t1) - b[d]
if (t1 < t) t = t1 <-- might be using the "image" cell here!
if (t > 0) d2 += t*t
3
return d2
```