計算天文学 II 第 10 回 データ構造とアルゴリズム (1)

牧野淳一郎

1 データ構造

計算機でプログラムを書く上で基本的なデータ構造というと

1. 配列
2. リスト
3. ツリー構造

といったものであり、基本的なアルゴリズムというと

1. サーチ
2. ソート
3. その他再帰的アルゴリズム

といったところになる。これらをまんべんなくカバーしようとするとそれだけでもう 1 学期必要になるので、今回からの講義では多少話が高度になるがツリー構造を使うすこし大がかりな計算法の話をして、基本的なアルゴリズムが実際にどのようにプログラムになるかを見ていくことにする。データ構造としては線形リストとツリー (木) 構造、アルゴリズムとしては主に再帰的アルゴリズムを扱う。

2 重力多体問題

対象となる問題は、重力多体問題である。これはつまり沢山の質点がお互いに重力で引き合って運動するというもので、星団、銀河等の基本的なモデルである。

運動方程式は

$$\frac{d^2\boldsymbol{x}_i}{dt^2} = -\sum_{j\neq i, 1\leq j\leq N} Gm_j \frac{\boldsymbol{x}_j - \boldsymbol{x}_i}{|\boldsymbol{x}_j - \boldsymbol{x}_i|^3}, \tag{1}$$

である。 $\boldsymbol{x}_i, m_i$ はそれぞれ粒子 i の位置、質量であり、G は重力定数である。

これをそのまま数値計算してもいいが、2 つの粒子が近付いた時に重力が発散するという問題がある。銀河のシミュレーションの場合には、実際の星の数は 10^{10} 個以上と非常に多いのに、それを多い場合でもせいぜい 100 万程度の数の粒子で表すので、粒子が重い分この問題がおきやすくなっている。この発散をまともに扱うには、粒子毎に時間刻みを変更するような特別な工夫が必要になる。

発散を避けるために、ポテンシャルとして純粋な $-Gm/r$ ではなくて、

$$\phi = -\frac{Gm}{\sqrt{r^2+\epsilon^2}} \tag{2}$$

というふうに $r=0$ で有限になるようにしたものを使うのが普通である。ϵ は定数で、計算結果に影響しないように十分小さくとる。

このようにポテンシャルをなまらしたときには、時間積分は簡単になって一定刻みでやればいい。

実際に沢山の粒子を使う上で問題になるのが、運動方程式の右辺の計算量である。一つの粒子への力を計算するのに他の全部からの力を計算しないといけないので、粒子が N 個あれば 1 ステップ当たりの計算量は N^2 になる。

これをもうちょっとうまくやろうというのが、次に説明するツリー法というものである。

3　ツリー法の原理

計算量を減らすには、遠くの粒子を適当にまとめたい。例えば遠くの方の粒子のかたまりをそれらの重心で置き換えてもいいし、高い精度が必要なら多重極展開も考えられる。遠くにいくに従ってかたまりを大きくしていけば、直観的には一つの粒子への力を $O(\log N)$ で計算でき、全粒子への力であればその N 倍の $O(N\log N)$ で計算できることになる。

もちろん、ここで問題なのはどうやってうまく粒子をかたまりに分けるかということである。ある粒子への力を計算するのに適した分け方を見つけるには、すくなくとも一度は全粒子をみないといけない。従って、その計算量は $O(N)$ より小さくはない。粒子毎に別の分け方をしたら、それだけで計算量が $O(N^2)$ になってしまう。

この問題は、すべての粒子に使える汎用の分け方というのをあらかじめ構成することができれば解決する。それをするのがツリー構造による階層的な空間分割ということになる。

図で説明しよう。3 次元の図は書きにくいし見る方も見にくいであろうから、2 次元で書くことにする (図 1)。まず、平面内の適当な領域に粒子が分布しているとしよう。粒子が有限個なので、そのすべてを含む正方形を考えることが出来る。すべて含んでいればよいので、別に最小である必要はないが、大きくても無駄なだけなので普通はそこそこ小さくとる。これを、まず 4 つの小正方形 (セル) にわける。で、そのそれぞれをさらに 4 つにわけるというのを再帰的に繰り返していく。

粒子が 0 個または 1 個しかなくなったところで止める。この空間 (平面) 分割に対応したツリー構造を考えると、根に全体の正方形に対応する節点があり、その下に 4 つの小正方形、さらにその下にそのそれぞれの中の 4 つ $\cdots$ と続いていって、ツリーの葉には粒子 1 つ 1 つがくることになる。この方法では、粒子の数密度が高いところでは細かく、そうでないところでは粗いという意味で、適応的な構造が実現されていることに注意してほしい。3 次元にすれば、正方形 4 個の代わりに立方体 8 個となるが、それ以外は全く同じである。

次に、どうやってこのツリー構造を使って一つの粒子への重力が計算できるかということを考えてみる。このためには、ツリーのある節点が表す立方体内の粒子全体からのある粒子への力を計算する方法を与えればよい。(なお、以下では、節点に対応する立方体、あるいはそのなかの粒子全体のことも単に節点ということがある) 系全体からの力は、単に根節点からの力として計算できる。一

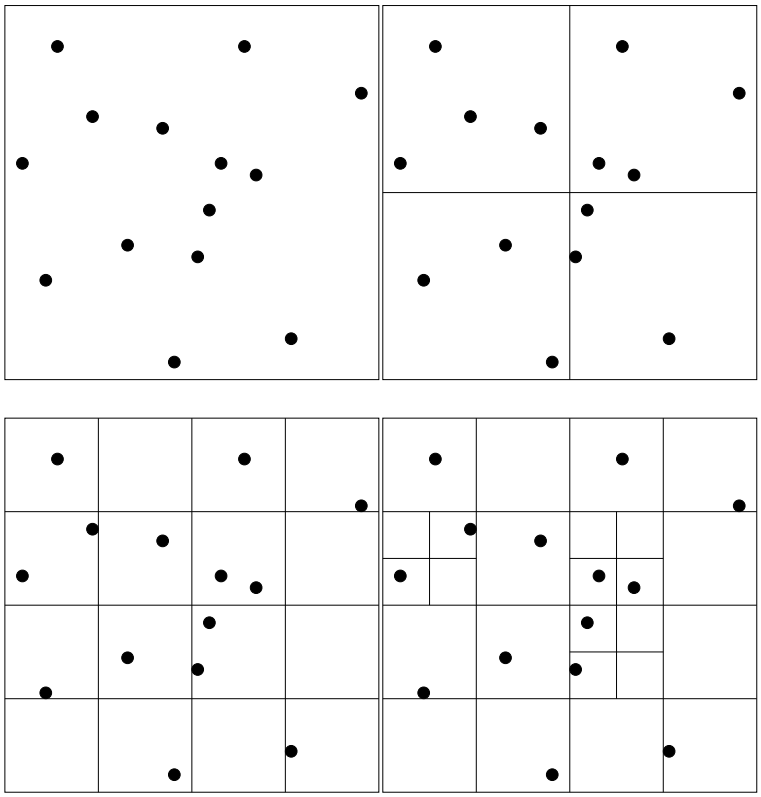

図 1: 4 分木の構築

つの節点からの力を以下のように計算することにする

$$\text{ある節点からある粒子への力} = \begin{cases} \text{その節点の重心からの力} \\ (\text{節点と粒子が「十分離れている」}) \\ \text{その節点の子節点からの力の合計} \\ (\text{それ以外}) \end{cases} \tag{3}$$

「十分離れている」かどうかの判定には通常は以下のような基準を使う。

$$\frac{l}{d} < \theta \tag{4}$$

ここで d は粒子と節点の重心の距離、 l は節点に対応する立方体の一辺の長さである。パラメータ θ は見込み角といわれる量であり、計算精度と計算量を制御する。高い精度を得たければ θ を小さくすればいいが、漸近的には θ^{-3} に比例して計算量が増える。計算精度をあげるもう一つの方法は、重心だけを考えるのでなく多重極展開を作っておくことである。

ツリー法を実際に使うには、各節点についてその中の粒子全体の重心の位置と質量 (あるいは多重極展開の展開係数) をあらかじめ求めておく必要がある。これらはやはり再帰的に定義することができる。つまり、子節点の中の粒子全体の重心の位置と質量がわかっていれば、親節点の情報は計算できる。多重極展開の展開係数についても (計算は繁雑であるが原理的には) おなじことである。

4　粒子データと線形リスト

というわけで、計算法をきわめて簡単に説明したが、プログラムとしてはどんなふうになるだろうか？以下、実際例を見ていくことにする。

実際のプログラムを書く関係上、どうしても準備が多くなるがまあちょっと我慢を。以下は、「粒子」のデータ構造である。

```
#ifndef  PARTICLE_H
#  define   PARTICLE_H
/*-----------------------------------------------------------------------------
 *  nbody-particle : basic class for simple nbody implementation
 *  J. Makino 1998/11/29
 *  Modified  2005/01/16
 *-----------------------------------------------------------------------------
 */
#include "myvector.h"

class particle
    {
    public:
        myvector pos;
        myvector vel;
        myvector acc;
        real phi;
        real mass;
        particle *  next;
        particle *  subnext;
        particle(){
            pos = 0.0;
            vel = 0.0;
            acc = 0.0;
            phi =  mass =  0.0;
        }
        void predict(real dt){
            real dt2 = dt*dt*0.5;
            pos = pos + dt*vel + dt2*acc;
            vel +=  (dt*0.5)*acc;
        }
        void correct(real dt){
            vel +=  (dt*0.5)*acc;
        }
};
#endif
```

これは、一応 C++ の「クラス」の文法には従っているが、実際には C の構造体であると思っていい。 private メンバはなくてすべてのデータメンバが外から見えるし、メンバー関数も初期設定以外にはなにも使っていないからである。

余談になるが、「粒子」クラスのような、物理的な実体とプログラムの中での構造がほぼ直接に対応するものについては、 C++ で一般的に使われるような大げさなデータ構造が適切なものであるかどうかはちょっと疑問である。

ここで、 myvector は前に使ったものとほぼ同じだが、3 次元用に書き下したものを使っている。

あ、それから、 next と subnext が、「粒子」クラスへのポインタになっている。ここでポインタというのは、単純には、ある変数の計算機メモリ内でのアドレスを保持するような変数である。 C/C++ の場合には、Fortran とは違ってポインタ変数を極めて柔軟に使うことができ、複雑なアルゴリズムやデータ構造を表現可能にすると同時にプログラムを難解なものにすることにもなっている。

4.1 ポインタとリスト

と、ごたくを並べてもしょうがない。簡単な例で説明してみよう。以下のようなコード断片を考えてみる。

```
particle a, b, c;
a.next = &b;
b.next = &c;
c.next = NULL;
for (particle * p = &a; p!= NULL; p = p->next){
    setup_particle(p);
    cout << p->pos << endl;
}
```

ここで、 & は、ある変数のアドレスを返す「演算子」であり、演算結果の型はその変数に対するポインタ型になる。逆に、 * はある変数へのポインタ型がさしているアドレスにあるもともとの変数そのものとして使える。

うーむ、いまいち意味がわかりませんね、、、上のコードが何をするかを見ていこう。まず、 a, b, c という particle 型の変数 3 個を宣言する。これらにはそれぞれどこか適当なメモリアドレスがコンパイラによって割り当てられるわけである。ここで、a.next = &b; とすると、next は particle 型へのポインタであったので、クラス変数 a のメンバ next の値が b のアドレスである、つまり a.next が b を指しているということになる。この状態では、例えば a.next->pos と b.pos は同じように b のメンバ pos になる。NULL は、「このポインタはどこも指していない」ということを示すために使われる特別な値である。

このように、ある構造体 (クラス) のメンバに、その構造体自体を指すポインタ変数を入れておくことで、構造体変数を数珠つなぎにしておいてずるずるたどっていくようなことができる。この、数珠つなぎ構造のことを、「線形リスト」という。

この例では配列とどう違うのかわからないが、主な利点は並べかえ (途中に新しい要素を入れたり、抜いたりする) 時の手間がデータの数によらないことである。欠点は、最初から順番に見ていくことしかできないことで、途中のデータをぱっと見たいというような時には具合がわるい。

このツリー法のプログラムでは、next によって作られる線形リスト構造を、ツリーの各セルの中に入っている粒子の集合を表現するのに使う。

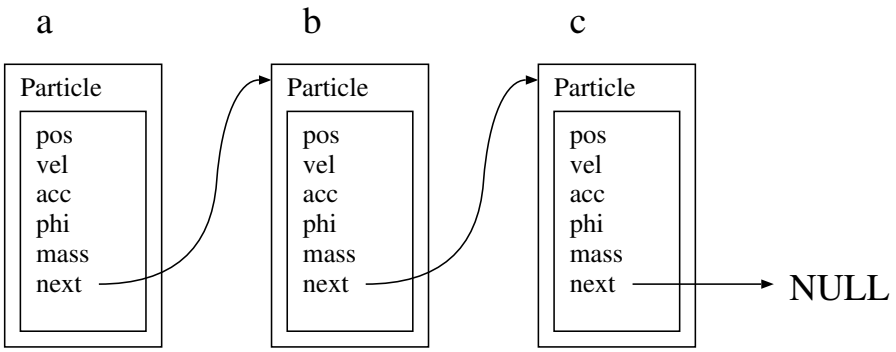

図 2: 粒子の線形リスト

詳しくはもうちょっとあとで。そのまえに、木構造の表現のほうにいこう。

5 木構造と再帰的アルゴリズム

5.1 ツリーのためのデータ構造

以下が、今回のプログラムで使う木構造のためのクラス定義である。

```
#ifndef  BHTREE_H
#  define   BHTREE_H
/*-------------------------------------------------------------------------------
 *  BHtree : basic class for C++ implementation of BH treecode
 *  J. Makino 2001/1/23
 *  Modified  2005/1/16
 *-------------------------------------------------------------------------------
 */


#include "myvector.h"
#include "particle.h"

class bhnode
{
private:
    myvector cpos;
    real l;
    bhnode * child[8];
    particle * pfirst;
    int nparticle;
    myvector pos;
    real mass;

public:
    bhnode(){
        cpos = 0.0;
        l = 0.0;
        for(int i = 0; i<8;i++)child[i] = NULL;
        pfirst = NULL;
```

```
        nparticle = 0;
        pos = 0.0;
        mass = 0.0;
    }
};
#endif
```

BH というのがやたらでてくるが、これはこのアルゴリズムを 1986 年に提唱した Barnes と Hut (当時はどちらもプリンストン高等研究所。今は Barnes はハワイ大学天文学科) の頭文字をとっているからである。 cpos は立方体の中心座標、 l は辺の長さである。 child は子セル (ノード) へのポインタの配列であり、最大 8 個の子セルがあるので要素数 8 の配列にしてある。これを使うことで、線形リストのように 1 つだけを指すのではなく、8 個の場所を指せるわけで、それによって木構造が表現できるわけである。

pfirst は中にある粒子のリストの先頭を指す。 nparticle はこのリストの長さである。

ここで、ちょっと注意して欲しいのは、長さは本来必要ではないということである。つまり、リストを作る時に、最後の要素が NULL を指しているというふうに決めたので、長さはリストをたどっていって数えればわかるわけで、本来長さを別に覚えておく必要はない。が、数えなくても長さがわかると便利なことが多いので別に覚えておくことにする。このようにした場合には、情報が冗長になっているので 2 つの情報が矛盾しないように、あるいはちゃんと意味があるほうを使うように注意する必要がある。

5.2 ツリーを作るアルゴリズム

実際に木を作る手順を見てみる。最初にすることは、1 つ立方体をつくって、それをルート (根) とし、全粒子をその中にあるということにする、具体的には、全粒子をルートの pfirst から始まるリストに入れておくことである。以下がそれを実現するプログラムになる。

```
void bhnode::assign_root(myvector root_pos, real length, particle * p, int np)
{
    pos = root_pos;
    l = length;
    pfirst = p;
    nparticle = np;
    for(int i=0;i<np-1;i++){
        p->next = p+1;
        p++;
    }
    p->next = NULL;
}
```

これはメンバ関数なので、bhnode クラス変数のメンバはメンバ名だけで使える。中心座標、辺の長さと、粒子の配列、粒子数が引数で渡ってくるので、中心座標、辺の長さをメンバ変数に代入し、pfirst から始まるリストを配列の要素を順にたどっていくように作っておく。

これは、元々の粒子の配列の順番にリストを作っただけで、この段階では意味がないように見える。何故わざわざこういうことをするかは次に木構造を作るところでわかる (と思う)。

次に実際に木構造を作っていく。これは、以下のようになる

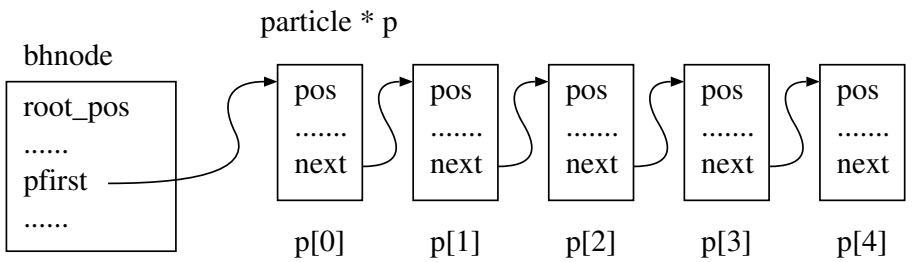

図 3: ルートノードに全粒子をいれた状態

```
void bhnode::create_tree_recursive(bhnode * & heap_top, int & heap_remainder)
{
    for(int i = 0;i<8;i++)child[i] = NULL;
    particle * p = pfirst;
    for(int i=0 ; i<nparticle; i++){
        particle * pnext= p->next;
        int subindex = childindex(p->pos,cpos);
        assign_child(subindex, heap_top, heap_remainder);
        child[subindex]->nparticle ++;
        p->next = child[subindex]->pfirst;
        child[subindex]->pfirst = p;
        p=pnext;
    }
    for(int i=0; i<8;i++) if(child[i]!=NULL){
        if (child[i]->nparticle > 1){
            child[i]->create_tree_recursive(heap_top, heap_remainder);
        }else{
            child[i]->pos = child[i]->pfirst->pos;
            child[i]->mass = child[i]->pfirst->mass;
        }
    }
}
```

ここで、渡ってきている引数は、実際には bhnode 構造体の配列の先頭アドレスと、その配列の要素数、というか正確にはその配列のどこか途中のアドレスと残りの要素数である。この配列は、木構造全部を作るのに十分な程度の大きさをとっておく。木に新しいノードを付け加えるためには配列の先頭から順番に要素を割り当てていく。

一般には、木構造のような動的な構造を作るには、 new 演算子を使って必要に応じて新しいメモリ領域を割り当てる。が、ここでは木構造は時間積分のステップ毎に新しく作るので、 new をその度に使ってはメモリが足りなくなるとかややこしい問題がある。そこで、自前でメモリを配列に確保しておいてそこにあるものを使うことにする。足りなくなったらここではエラーを出して終る。

まず、最初にすることは、自分の下にある全粒子をたどっていって、それがどの子ノードにいくかを判定し、結果に応じて対応するリストに挿入 (先頭に追加) することである。ここで、 next は自分のリストなので、これを書き変えるとその後がたどれなくなる。このため、ループの最初に自分の次というのを pnext に憶えておく。判定は、各座標方向に対して、中心より上か下かを見るだけである。これは、以下の関数が行う。

```
int childindex(myvector pos, myvector cpos)
{
    int subindex=0;
    for(int k=0;k<3;k++){
        subindex <<= 1;
        if (pos[k] > cpos[k])subindex += 1;
    }
    return subindex;
}
```

どの子ノードに入るかが決まったら、実際に子ノードに入れる。その前に、その子ノードに bhnode を割り当てたかどうかをチェックし、既に割り当ててなかったら新しく割り当てる。これは以下の関数である。

```
void bhnode::assign_child(int subindex,
                          bhnode * & heap_top,
                          int & heap_remainder)
{
    if (heap_remainder <= 0){
        cerr << "create_tree: no more free node... exit\n";
        exit(1);
    }
    if (child[subindex]==NULL){
        child[subindex] = heap_top;
        heap_top ++;
        heap_remainder -- ;
        child[subindex]->cpos = cpos + myvector( ((subindex&4)*0.5-1)*l/4,
                                                 ((subindex&2)    -1)*l/4,
                                                 ((subindex&1)*2  -1)*l/4);
        child[subindex]->l = l*0.5;
        child[subindex]->nparticle=0;
    }
}
```

ここでは、座標の計算とかもついでにしているので少し長い関数になっている。

実際の粒子の割り当ては、

```
        child[subindex]->nparticle ++;
        p->next = child[subindex]->pfirst;
        child[subindex]->pfirst = p;
```

の部分である。子ノードの粒子の数を憶えておく nparticle を 1 増やし、child[subindex]->pfirst を p にすればいいわけだが、そうするとこれまで child[subindex]->pfirst が指していたものがどこかにいってしまうので、その前に child[subindex]->pfirst を p->next に代入しておく。

全部の粒子の子ノードへの振り分けがすんだら、粒子が 2 つ以上入っている子ノードをさらに分割する。これは、ここではこの関数自身を再帰的に呼び出すことで実現されている。粒子が 1 つだけなら、その質量、位置を木構造データのほうの変数にコピーしておく。

ここで、再帰的な関数が、「自分のリストを見て子供のリストを作る」という形になっているので、最初にルートノードに対してリストを作る必要があったことがわかる。

と、今日はとりあえずこれくらいで。来週はこれを使った重力の計算をしてみる。

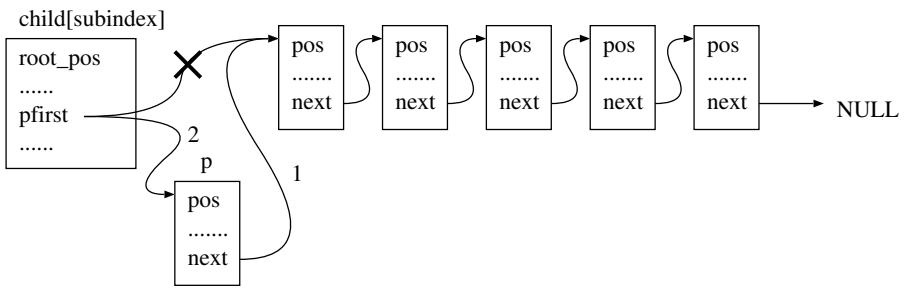

図 4: 粒子リストに粒子を挿入

6 各関数のテスト

というわけで、一応木構造をつくるためのデータ構造、アルゴリズムとその実現を紹介した。これを実際に多体問題の数値解を求めるプログラムにするためには、ノードの物理データの計算、重力計算、時間積分、初期条件の生成 / 読み込み、結果の表示 / 解析等いろんなことをするプログラムを作る必要がある。

それらに入る前に、とりあえず今まで出来たところを動かしてみることにする。

割合いろんな関数を作っていろんなことをやらせるので、全体を動かして時にちゃんと動いているかどうかをどうやって判断し、さらに動いていない時にどこがおかしいかを見つけ出すのはそんなに簡単ではなくなる。

ある程度大きな、沢山の関数からできたプログラムをテスト・デバッグするための基本的な考え方は以下の 2 つである。

1. それぞれの関数を単体でテストする。
2. 組み合わせたプログラムをテストするが、動作が正しいかどうかを関数ごとにチェックする。

もちろん、プログラムをデバッグする時の大きな問題は、

- プログラムは書いた人の意図した通りにではなく、書いた通りに動く
- プログラムを書いた人の意図が正しいとは限らない

ということである。特に後者がデバッグを難しいものにする。

というわけで、まず、ツリー構造を作り、それが正しいかどうか調べる。メインプログラムは以下のような感じになる。

```
#ifdef TREETEST
int main()
{
    particle * pp;
    int n;
    cerr << "Enter n:";
    cin >> n ;
    pp = new particle[n];
```

```
    double rsize = 1.0;
    create_uniform_sphere(pp, n, 0 , rsize);
    for(int i =0;i <n;i++){
        PRC(i); PRL(pp[i].pos);
    }
    bhnode * bn = NULL;

    int nnodes = n*2;
    bn = new bhnode[nnodes];
    bn->assign_root(myvector(0.0), rsize*2, pp, n);
    int heap_remainder = nnodes-1;
    bhnode * btmp = bn + 1;
    bn->create_tree_recursive(btmp,  heap_remainder);
    PRL(bn->sanity_check());
    bn->dumptree();
    return 0;
}
#endif
```

プログラム全体は

http://grape.astron.s.u-tokyo.ac.jp/%7Emakino/kougi/keisan_tenmongakuII/programs/simpletree/B

にあるので細かい関数については省略するが、関数 sanity_check はちょっと中を見ておこう。

```
int bhnode::sanity_check()
{
    int i;
    int iret = 0;
    if (nparticle == 1 ){
        // this is the lowest level node. Things to check:
        // particle is in the cell
        particle * p = pfirst;
        if(inbox(pos,p->pos,l)){
            cerr << "Error, particle out of box ... \n";
            dump();
            return 1;
        }
    }else{

        // This is the non-leaf node. Check the position and side
        // length of the child cells and then check recursively..
        for(i=0;i<8;i++){
            if (child[i] != NULL){
                int err = 0;
                err = child[i]->sanity_check();
                if (l*0.5 != child[i]->l) err += 2;
                myvector relpos = cpos-child[i]->cpos;
                for (int k = 0 ; k<ndim;k++){
                    if (fabs(relpos[k]) !=l*0.25)err += 4;
                }
                if (err){
                    cerr << "Child " << i << " Error type = " << err << endl;
                    dump();
                }
                iret += err;
            }
        }
    }
    return iret;
}
```

ここでは、以下のようなチェックを行なっている。

1. ノードに粒子が 1 つだけなら、それがちゃんと中にあるかどうか。
2. そうでなければ、自分の子供は正しい位置、大きさを持つかどうか。

で、さらに再帰的に子供についても同じチェックをする。

inbox はこんなものである。

```
int inbox(myvector  & cpos, // center of the box
          myvector  & pos,  // position of the particle
          real l)         // length of one side of the box

{
    for(int  i = 0; i< ndim; i++){
        if (fabs(pos[i]-cpos[i]) > l*0.5) return 1;
    }
    return 0;
}
```

ここではベクトルの各要素についてそれぞれ調べる必要がある。

さて、この関数が「エラー」を返して来たとしても、少なくとも以下の 4 通りの可能性がある。

- ツリーを作る関数が間違っている。
- sanity_check が間違っている。
- ツリーを作る以前に既になにか間違っている。
- 上の 3 つの任意の組み合わせ。

というわけで、最初は粒子 1 つとか 2 つで何ができるかみるといったところから始めないと後が大変ではある。逆に数が少なければいろいろ書き出して人間がチェックするのも不可能ではない。以下の関数はツリー構造そのものを再帰的にプリントアウトする。 spc は単に引数の数だけ空白文字を印刷する関数である。

```
void bhnode::dumptree(int indent)
{
    int i;
    spc(indent); cerr << "node center pos " << cpos ;
    if (nparticle == 1){
        cerr << " IS LEAF" ;PRL(nparticle);
        particle * p = pfirst;
        for(i = 0; i < nparticle; i++){
            for(int j=0;j<indent+2;j++)cerr << " ";
            PRL(p->pos);
            p= p->next;
        }
    }else{
        cerr << " IS _not_ LEAF ";PRL(nparticle);
```

```
        for(i=0;i<8;i++){
            if (child[i] != NULL){
                child[i]->dumptree(indent + 2);
            }
        }
    }
}
```

以下は $N = 4$ での実行結果である

```
simpletree>BHtreetest
Enter n:4
nbody = 4,  power_index = 0
i = 0,  pp[i].pos = -0.815405  -0.0255656  0.0535006
i = 1,  pp[i].pos = -0.0911332  -0.533643  0.662584
i = 2,  pp[i].pos = 0.863463  0.136119  0.112189
i = 3,  pp[i].pos = -0.49528  -0.403606  0.751962
bn->sanity_check() = 0
node center pos 0  0  0 IS _not_ LEAF nparticle = 4
  node center pos -0.5  -0.5  0.5 IS _not_ LEAF nparticle = 3
    node center pos -0.75  -0.25  0.25 IS LEAFnparticle = 1
      p->pos = -0.815405  -0.0255656  0.0535006
    node center pos -0.25  -0.75  0.75 IS LEAFnparticle = 1
      p->pos = -0.0911332  -0.533643  0.662584
    node center pos -0.25  -0.25  0.75 IS LEAFnparticle = 1
      p->pos = -0.49528  -0.403606  0.751962
  node center pos 0.5  0.5  0.5 IS LEAFnparticle = 1
    p->pos = 0.863463  0.136119  0.112189
```

それらしいツリー構造ができていることがわかる。

来週は、力の計算とか、残りのところをプログラムする。