

Algoritmos e Estrutura de Dados

Relatório do trabalho sobre Binary Tree’s

16/12/2018

Bernardo Rodrigues nº Mec: 88835 (50%)

Vinícius Benite Ribeiro nº Mec: 82773 (50%)

**Introdução**

Para a disciplina de Algoritmos e Estrutura de Dados foi-nos proposto o estudo de árvores binárias ordenadas com n elementos, nomeadamente a média da altura da árvore, o número médio de folhas, e o custo médio de encontrar elementos na árvore, quer estejam presentes ou não.

Para tal desenvolvemos funções na linguagem C que juntamente com um programa previamente oferecido pelo professor da cadeira, cria várias árvores binárias de tamanho crescente até 10000 elementos, calcula o seu tamanho, número de folhas e verifica se alguns números pertencem à árvore e devolve a média de tempo que cada número demorou a procurar.

Para criar as árvores o programa insere os números 1,3,5,7…, 2n-1 por ordem aleatória, sendo n o número de elementos da árvore.

**Count leaves**

Nas árvores binárias, as folhas são os nodes cujos ambos os ponteiros (ponteiro para a esquerda e ponteiro para a direita) são nulos.

O número de folhas é calculado recursivamente através da função count\_leaves, quando esta é chamada verifica se o ponteiro para a esquerda ou o ponteiro para a direita do node onde está são não nulos, caso não sejam ela é invocada nos nodes seguintes. Quando chega a um node com ponteiros nulos, acrescenta 1 ao número de folhas e no final devolve o resultado.

Seguidamente, o programa calcula o mínimo, o máximo, a média e a standard deviation/variance (std) de folhas das árvores geradas aleatoriamente. O número de folhas nunca é maior que o número de elementos.

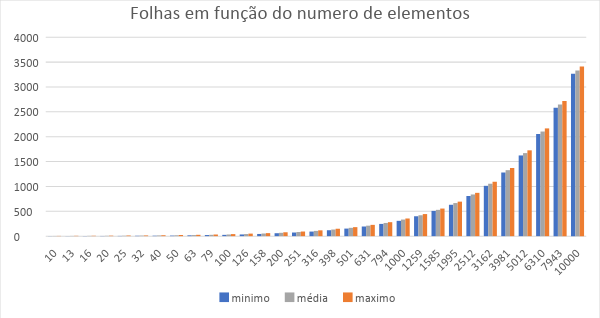
O número de folhas possíveis é entre 1, se todos os nodes da árvores tiverem apenas um node filho, e 2h-1 sendo h a altura da árvores

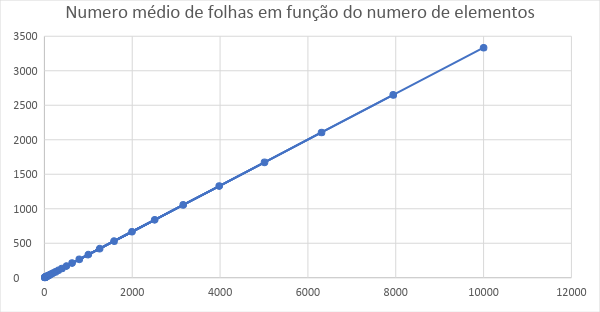
Nº folhas ∈ [1, 2h-1 ]

Numa árvores cheia(full), ou seja, que todos os nodes têm 0 ou 2 nodes filhos o número de folhas é igual ao número de nodes com 2 nodes filhos + 1 ou (n+1)/2.

**Dados para o número de folhas**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | min | max | Mean | std |
| 10 | 2 | 5 | 3.6540 | 0.6843 |
| 13 | 2 | 7 | 4.6940 | 0.7631 |
| 16 | 3 | 8 | 5.6700 | 0.8585 |
| 20 | 4 | 10 | 6.9570 | 0.9824 |
| 25 | 5 | 13 | 8.6460 | 1.1174 |
| 32 | 8 | 14 | 11.0430 | 1.2013 |
| 40 | 9 | 18 | 13.6440 | 1.3451 |
| 50 | 11 | 23 | 16.9760 | 1.5399 |
| 63 | 16 | 27 | 21.3560 | 1.6998 |
| 79 | 22 | 33 | 26.7660 | 1.8053 |
| 100 | 26 | 41 | 33.6570 | 2.1188 |
| 126 | 34 | 51 | 42.3280 | 2.3559 |
| 158 | 44 | 61 | 53.0230 | 2.7067 |
| 200 | 59 | 78 | 67.1120 | 3.0538 |
| 251 | 73 | 93 | 83.7730 | 3.3999 |
| 316 | 92 | 117 | 105.6710 | 3.6566 |
| 398 | 119 | 149 | 133.0730 | 4.3282 |
| 501 | 152 | 181 | 167.4630 | 4.8607 |
| 631 | 194 | 227 | 210.8870 | 5.4547 |
| 794 | 246 | 281 | 265.1700 | 5.8470 |
| 1000 | 308 | 355 | 333.6360 | 6.5544 |
| 1259 | 400 | 444 | 420.0940 | 7.4120 |
| 1585 | 508 | 555 | 529.0860 | 7.9671 |
| 1995 | 631 | 694 | 665.4450 | 9.4760 |
| 2512 | 807 | 871 | 838.3090 | 10.4556 |
| 3162 | 1012 | 1094 | 1054.1410 | 11.8696 |
| 3981 | 1280 | 1371 | 1326.9490 | 13.3436 |
| 5012 | 1623 | 1726 | 1670.1980 | 15.0302 |
| 6310 | 2054 | 2167 | 2104.0740 | 16.4703 |
| 7943 | 2583 | 2720 | 2647.9980 | 18.9578 |
| 10000 | 3267 | 3411 | 3332.4030 | 20.8589 |





Pelo gráfico podemos ver que o número médio de folhas cresce linearmente com o número de elementos no gráfico. Porém, os valores obtidos são apenas uma média e devido às propriedades dos gráficos, um gráfico pode ter mais nodes e mesmo assim ter menos folhas que um outro com menos nodes e quanto maior o número de nodes maior a probabilidade de tal acontecer, visto que a variância aumenta ao mesmo tempo que o nº de nodes aumenta.

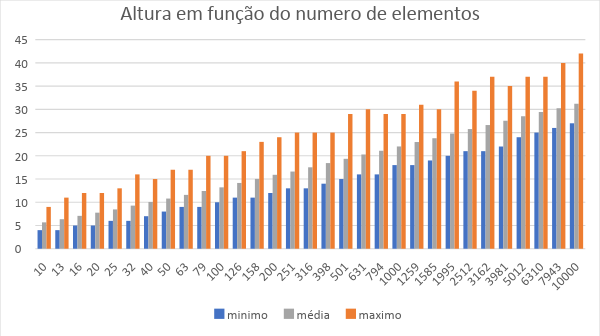
**Tree height**

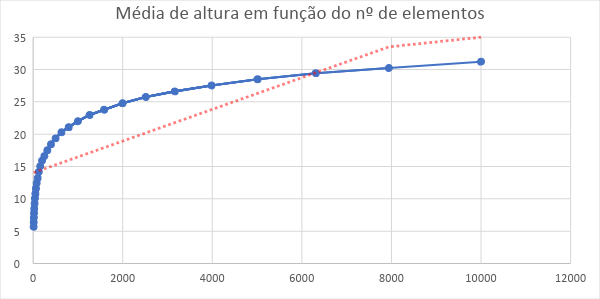
A altura de uma árvores é o número de nível mais alto da árvores, um nível é contado sempre que cada node tem pelo menos um ponteiro não nulo, sendo a raiz considerada normalmente como nível 0. Para o cálculo da altura das árvores é usada a função tree\_height que subdivide a árvore em várias árvores mais pequenas. Começando por calcular a altura da árvore da esquerda do node raiz e depois a altura da árvore da direita do node, retornando o valor mais alto. Sempre que um node é nulo a função retorna 0, pois não se pode contar o node não existente para a altura da árvore.

O programa calcula também o mínimo, máximo, média e variância da altura. A altura máxima seria num caso degenerado em que cada node teria um filho e a altura seria igual a N. Para qualquer árvore binária a altura mínima é log2 ( N ). Uma árvore binária com L folhas tem pelo menos log2( L ) + 1 níveis.

**Dados para a altura das árvores**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| n | min | max | mean | std |
| 10 | 4 | 9 | 5.6750 | 0.8885 |
| 13 | 4 | 11 | 6.3510 | 0.9828 |
| 16 | 5 | 12 | 7.0760 | 1.1438 |
| 20 | 5 | 12 | 7.7630 | 1.1759 |
| 25 | 6 | 13 | 8.4590 | 1.2338 |
| 32 | 6 | 16 | 9.2800 | 1.3075 |
| 40 | 7 | 15 | 10.0680 | 1.3976 |
| 50 | 8 | 17 | 10.7980 | 1.4047 |
| 63 | 9 | 17 | 11.5900 | 1.4470 |
| 79 | 9 | 20 | 12.4260 | 1.5371 |
| 100 | 10 | 20 | 13.2110 | 1.4834 |
| 126 | 11 | 21 | 14.1590 | 1.6192 |
| 158 | 11 | 23 | 15.0280 | 1.7231 |
| 200 | 12 | 24 | 15.9160 | 1.7941 |
| 251 | 13 | 25 | 16.6120 | 1.6809 |
| 316 | 13 | 25 | 17.5270 | 1.6880 |
| 398 | 14 | 25 | 18.4350 | 1.8253 |
| 501 | 15 | 29 | 19.3570 | 1.8203 |
| 631 | 16 | 30 | 20.2880 | 1.9378 |
| 794 | 16 | 29 | 21.0880 | 1.8773 |
| 1000 | 18 | 29 | 22.0120 | 1.8498 |
| 1259 | 18 | 31 | 22.9660 | 1.9165 |
| 1585 | 19 | 30 | 23.7820 | 1.8790 |
| 1995 | 20 | 36 | 24.7970 | 2.0287 |
| 2512 | 21 | 34 | 25.7570 | 2.0420 |
| 3162 | 21 | 37 | 26.6200 | 2.0469 |
| 3981 | 22 | 35 | 27.5410 | 2.0155 |
| 5012 | 24 | 37 | 28.4900 | 2.0596 |
| 6310 | 25 | 37 | 29.4390 | 2.1578 |
| 7943 | 26 | 40 | 30.2350 | 2.1203 |
| 10000 | 27 | 42 | 31.2180 | 2.0670 |





Vemos que a média da altura das árvores cresce com base na função logarítmica, isto deve-se à árvore estar organizada e os números gerados aleatoriamente não distam muito entre si, o que faz com vários níveis da árvore estejam preenchidos.

A variância também não cresce não muito, comparativamente à variância do número de folhas.

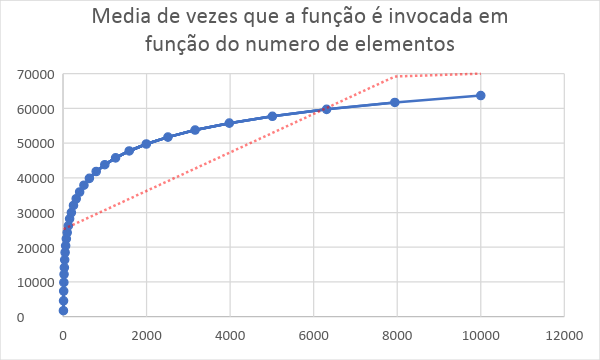
**Functions calls on hit**

Para calcular o custo de encontrar um elemento pertencente à árvore implementamos a função functions calls on hit que calcula quantas vezes é invocada a função para procurar um elemento da árvore.

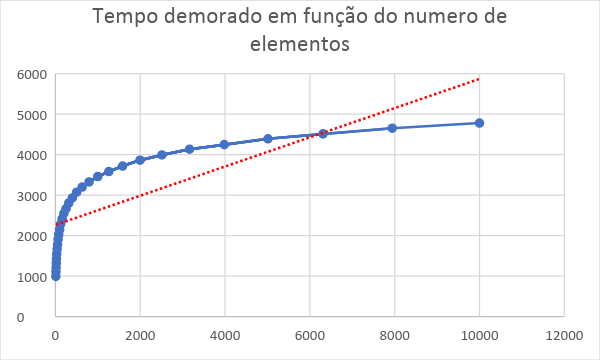
**Dados para function calls on hit**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| n | mean | std |  | n | mean | std |
| 10 | 1727,9079 | 995,3592 |  | 398 | 35910,2473 | 2931,1452 |
| 13 | 4573,8605 | 1106,3924 |  | 501 | 37868,7756 | 3072,8061 |
| 16 | 7368,0244 | 1212,9399 |  | 631 | 39835,3319 | 3200,0696 |
| 20 | 9862,6388 | 1319,483 |  | 794 | 41817,4164 | 3325,8279 |
| 25 | 12188,927 | 1428,3338 |  | 1000 | 43772,0618 | 3460,972 |
| 32 | 14136,2182 | 1547,9115 |  | 1259 | 45742,8441 | 3582,4391 |
| 40 | 16362,6321 | 1670,4442 |  | 1585 | 47727,7485 | 3718,8585 |
| 50 | 18487,089 | 1780,6916 |  | 1995 | 49736,0428 | 3863,6593 |
| 63 | 20422,8532 | 1906,3131 |  | 2512 | 51721,2077 | 3992,5758 |
| 79 | 22432,158 | 2028,914 |  | 3162 | 53750,0568 | 4134,6306 |
| 100 | 24217,6795 | 2152,2804 |  | 3981 | 55736,1395 | 4246,6413 |
| 126 | 26149,9565 | 2291,4393 |  | 5012 | 57717,8177 | 4390,8675 |
| 158 | 28199,9588 | 2416,1549 |  | 6310 | 59710,9548 | 4511,9743 |
| 200 | 29992,9779 | 2549,5497 |  | 7943 | 61708,6284 | 4653,1345 |
| 251 | 32029,2524 | 2664,9504 |  | 10000 | 63686,1239 | 4780,5411 |
| 316 | 33967,3905 | 2806,2587 |  |

Os valores da média são os valores da soma de média de 1000 árvores, para a média de cada árvore basta dividir o valor da média por 1000



Como a árvore está organizada a pesquisa por um elemento da árvore vai ser facilitada, pois na pesquisa apenas basta verificar se o valor do node onde nos encontramos é superior ou não ao valor pretendido, caso sim avançar para o node da esquerda(ou direita, dependendo de como a árvore está organizada), caso não avançar para o contrário e repetir até encontrar o valor



O tempo tal como a média de vezes que a função é invocada cresce com base na função logarítmica, pelas mesmas razões apresentadas para a média.

**Código do programa**

//

// Tomás Oliveira e Silva, AED, November 2018

//

// Bernardo Rodrigues

// n\_Mec = 88835

// Vinicius Benite Ribeiro

// n\_Mec = 82773

//

// empirical study of random ordered binary trees

//

#include <math.h>

#include <stdio.h>

#include <stdlib.h>

#include "elapsed\_time.h"

//

// each node of our ordered binary tree will store a long integer

//

// the root of the tree should be declared as follows (set initially to an empty tree):

//

// tree\_node \*root = NULL;

//

typedef struct tree\_node

{

struct tree\_node \*left; // pointer to the left branch (a sub-tree)

struct tree\_node \*right; // pointer to the right branch (a sub-tree)

struct tree\_node \*parent; // pointer to the parent node (NULL for the root of the tree)

long data; // the data item (we use a long here)

}

tree\_node;

//

// insert a node in the tree (it is assumed that the tree does not store repeated data)

//

// use it as follows (example):

//

// insert\_node(&root,&new\_node);

//

static void insert\_node(tree\_node \*\*link,tree\_node \*n)

{

tree\_node \*parent;

parent = NULL;

while(\*link != NULL)

{

if(n->data == (\*link)->data)

{

fprintf(stderr,"insert\_node: %ld is already in the tree\n",n->data);

exit(1);

}

parent = \*link;

link = (n->data < (\*link)->data) ? &((\*link)->left) : &((\*link)->right); // select branch

}

\*link = n;

n->parent = parent;

n->left = n->right = NULL;

}

//

// count the number of leaves of the tree

//

// use if as follows (example):

//

// int n\_leaves = count\_leaves(root);

//

static int count\_leaves(tree\_node \*link)

{

// TO DO: delete the next line of code and place your code here

if (link == NULL) { //Checar se a arvore esta vazia

return 0;

}

if (link->left == NULL && link->right == NULL) {

return 1;

}

else

{

return count\_leaves(link->left)+count\_leaves(link->right);

}

}

//

// compute the height of the tree

//

// use if as follows (example):

//

// int height = tree\_height(root);

//

static int tree\_height(tree\_node \*link)

{

// TO DO: delete the next line of code and place your code here

if (link == NULL) {

return 0;

}

else

{

int left\_height = tree\_height(link -> left); // Calcular a altura de cada sub arvore

int right\_height = tree\_height(link -> right);

if (left\_height > right\_height)

{ // Checar qual o maior

return (left\_height + 1);

}

else

{

return (right\_height + 1);

}

}

}

//

// recursive function used to search for the location of a data item

//

// use if as follows (example):

//

// tree\_node \*node = search\_tree(root,data);

//

static int search\_counter;

tree\_node \*search\_tree(tree\_node \*link,long data)

{

search\_counter++;

if(link == NULL) {

return NULL;

}

if(link->data == data)

return link;

return search\_tree((data < link->data) ? link->left : link->right,data);

}

//

// assuming that each data item is searched for with equal probability, compute the average number

// of recursive function calls to the search\_tree() function when

// 1) the search is successful (a hit)

// 2) the search is not successful (a miss)

//

// use them as follows (example):

//

// double average\_calls\_on\_hit = (double)count\_function\_calls\_on\_hit(root,0) / (double)number\_of\_nodes;

// double average\_calls\_on\_miss = (double)count\_function\_calls\_on\_miss(root,0) / (double)number\_of\_nodes;

//

static int hit;

static int count\_function\_calls\_on\_hit(tree\_node \*link,int level)

{

// TO DO: delete the next line of code and place your code here

if (link == NULL)

{

return 0;

}

if (search\_tree(link, link->data) != NULL) // Caso o dado seja encontrado, o custo sera a

{ // altura da arvore, no caso: level + 1

hit += level+1;

}

if (link->left && link->right) // Funciona recursivamente. Checa se

{ // o node possui filhos e desce 1 level

count\_function\_calls\_on\_hit(link->left,level+1);

count\_function\_calls\_on\_hit(link->right,level+1);

}

else if (!link->right)

{

count\_function\_calls\_on\_hit(link->left,level+1);

}

else if (!link->left)

{

count\_function\_calls\_on\_hit(link->right,level+1);

}

search\_counter--; // Como estamos chamando a funcao search\_tree, o search\_counter

return hit; // e incrementado erroneamente, logo, temos que subtrair a cada

} // nova chama da funcao.

static int count\_function\_calls\_on\_miss(tree\_node \*link,int level)

{

// TO DO: delete the next line of code and place your code here

if (link == NULL)

{

return 0;

}

return 0;

}

//

// random permutation of the n numbers 1, 3, 5, ..., 2\*n-1

//

// use if as follows (example):

//

// int n = 100;

// int a[n];

// rand\_perm(n,&a[0]);

//

static void rand\_perm(int n,int \*a)

{

int i,j,k;

for(i = 0;i < n;i++)

a[i] = 2 \* i + 1;

for(i = n - 1;i > 0;i--)

{

j = (int)floor((double)(i + 1) \* (double)rand() / (1.0 + (double)RAND\_MAX)); // range 0..i

k = a[i];

a[i] = a[j];

a[j] = k;

}

}

//

// main program

//

int main(int argc,char \*\*argv)

{

int details = (argc == 3 && argv[1][0] == '-' && argv[1][1] == 'a' && atoi(argv[2]) > 0) ? 1 : 0;

int n\_experiments = 1000; // TO DO: use more (1000000 should take 2 to 3 hours)

srandom(1u); // ensure reproducible results

printf(" data for %d random trees\n",n\_experiments);

printf(" maximum tree height number of leaves calls on hit calls on miss\n");

printf(" ---------------------- ------------------------------- -------------- --------------\n");

printf(" n min max mean std min max mean std mean std mean std\n");

printf("------ --- --- ------- ------ ----- ----- ---------- -------- ------- ------ ------- ------\n");

for(int n\_log = 1 \* 10;n\_log <= 4 \* 10;n\_log++)

{

int n = (int)round(pow(10.0,(double)n\_log / 10.0)); // the number of nodes of the tree

int a[n]; // the nodes' data

tree\_node \*root,nodes[n]; // the root and the storage space for the nodes of the tree

int h\_height[n + 1]; // for an histogram of the heights of the random trees

int h\_leaves[n + 1]; // for an histogram of the number of leaves of the random trees

double mean,std; // for mean and standard deviation computations

double x,hit[2],miss[2]; // for the average number of hits and misses

int m,M; // location of minima and maxima

printf("%6d",n);

//

// the example in the slides

//

if(n == 10)

{

root = NULL;

nodes[0].data = 3l; insert\_node(&root,&nodes[0]);

nodes[1].data = 1l; insert\_node(&root,&nodes[1]);

nodes[2].data = 9l; insert\_node(&root,&nodes[2]);

nodes[3].data = 7l; insert\_node(&root,&nodes[3]);

nodes[4].data = 5l; insert\_node(&root,&nodes[4]);

if(count\_leaves(root) != 2)

{

fprintf(stderr,"count\_leaves() returned a wrong value\n");

exit(1);

}

if(tree\_height(root) != 4)

{

fprintf(stderr,"tree\_height() returned a wrong value\n");

exit(1);

}

search\_counter = 0;

for(int i = 1;i <= 9;i += 2)

if(search\_tree(root,(long)i) == NULL)

return 1; // impossible if the program is correct

if(count\_function\_calls\_on\_hit(root,0) != search\_counter)

{

fprintf(stderr,"count\_function\_calls\_of\_hit() returned a wrong value\n");

exit(1);

}

search\_counter = 0;

for(int i = 0;i <= 10;i += 2)

if(search\_tree(root,(long)i) != NULL)

return 1; // impossible if the program is correct

printf("\nSC befoore count: %d\n", search\_counter);

if(count\_function\_calls\_on\_miss(root,0) != search\_counter)

{

printf("miss: %d\n", errou);

printf("\nSC after count: %d\n", search\_counter);

fprintf(stderr,"count\_function\_calls\_of\_miss() returned a wrong value\n");

exit(1);

}

}

//

// the experiments

//

for(int i = 0;i <= n;i++)

h\_height[i] = h\_leaves[i] = 0;

hit[0] = hit[1] = miss[0] = miss[1] = 0.0;

for(int n\_experiment = 0;n\_experiment < n\_experiments;n\_experiment++)

{

rand\_perm(n,&a[0]);

root = NULL;

for(int i = 0;i < n;i++)

{

nodes[i].data = (long)a[i];

insert\_node(&root,&nodes[i]);

}

h\_height[tree\_height(root)]++;

h\_leaves[count\_leaves(root)]++;

x = (double)count\_function\_calls\_on\_hit(root,0) / (double)n; // there are n nodes

hit[0] += x;

hit[1] += x \* x;

x = (double)count\_function\_calls\_on\_miss(root,0) / (double)(n + 1); // there are n+1 NULL pointers

miss[0] += x;

miss[1] += x \* x;

}

//

// output summary

//

mean = std = 0.0;

m = n + 1;

M = -1;

for(int i = 0;i <= n;i++)

if(h\_height[i] != 0)

{

mean += (double)i \* (double)h\_height[i];

std += (double)i \* (double)i \* (double)h\_height[i];

if(i < m) m = i;

if(i > M) M = i;

}

mean /= (double)n\_experiments;

std /= (double)n\_experiments;

std = sqrt(std - mean \* mean);

printf(" %3d %3d %7.4f %6.4f",m,M,mean,std);

mean = std = 0.0;

m = n + 1;

M = -1;

for(int i = 0;i <= n;i++)

if(h\_leaves[i] != 0)

{

mean += (double)i \* (double)h\_leaves[i];

std += (double)i \* (double)i \* (double)h\_leaves[i];

if(i < m) m = i;

if(i > M) M = i;

}

mean /= (double)n\_experiments;

std /= (double)n\_experiments;

std = sqrt(std - mean \* mean);

printf(" %5d %5d %10.4f %8.4f",m,M,mean,std);

mean = hit[0] / (double)n\_experiments;

std = hit[1] / (double)n\_experiments;

std = sqrt(std - mean \* mean);

printf(" %7.4f %6.4f",mean,std);

mean = miss[0] / (double)n\_experiments;

std = miss[1] / (double)n\_experiments;

std = sqrt(std - mean \* mean);

printf(" %7.4f %6.4f",mean,std);

printf("\n");

//

// output the tree height data

//

if(details != 0 || n == atoi(argv[2]))

{

printf(" i frac height\n");

printf(" ----- -----------\n");

for(int i = 0;i <= n;i++)

if(h\_height[i] != 0)

printf(" %5d %11.9f\n",i,(double)h\_height[i] / (double)n\_experiments);

printf(" ----- -----------\n");

}

//

// output the number of leaves data

//

if(details != 0 || n == atoi(argv[2]))

{

printf(" i frac leaves\n");

printf(" ----- -----------\n");

for(int i = 0;i <= n;i++)

if(h\_leaves[i] != 0)

printf(" %5d %11.9f\n",i,(double)h\_leaves[i] / (double)n\_experiments);

printf(" ----- -----------\n");

}

//

// done

//

fflush(stdout);

}

printf("------ --- --- ------- ------ ----- ----- ---------- -------- ------- ------ ------- ------\n");

printf("done in %.1f seconds\n",elapsed\_time());

return 0;

}