# Simulation Expirment for Proofing the Theoretical Assumption of Time Complexity for Binary Search Tree

**Muna M. Salih**
Technical College of Management/Baghdad
munaouceal@yahoo.co.uk

**Received in:28 April 2013 , Accepted in 14 December 2013**

## Abstract

It is frequently asserted that an advantage of a binary search tree implementation of a set over linked list implementation is that for reasonably well balanced binary search trees the average search time (to discover whether or not a particular element is present in the set) is O(log N) to the base 2 where N is the number of element in the set (the size of the tree).

This paper presents an experiment for measuring and comparing the obtained binary search tree time with the expected time (theoretical), this experiment proved the correctness of the hypothesis, the experiment is carried out using a program in turbo Pascal with recursion technique implementation and a statistical method to prove the above hypothesis. Search time is estimated by the number of comparisons needed.

**Keywords**: Time Complexity, Binary Search Tree, Big O Notation, Recursive Method, Chi square.

## Introduction

A binary tree is a special kind of tree, one that limits each node to no more than two children. A binary search tree, or BST, is a binary tree whose nodes are arranged such that for every node N, all of the nodes in N's left subtree have a value less than N, and all nodes in N's right subtree have a value greater than N. In the average case BSTs offer log (N) to the base 2 asymptotic time for inserts, deletes, and searches[1].

Binary search tree is a fundamental technique of programming, the advantage of using a binary search tree over an array is that a tree enable search, insertion, deletion operation to be performed efficiently [2].

Using simple calculation we can see that O(logN) is better than O(N) (the search time for an array). (In computer science, big O notation is used to classify algorithm by how they respond (e.g. in their processing time or working space requirements) to changes in input size)[3].

So let's say that we have 8 items that we place into an array and a binary tree. In the array, search time will be O (8) = 8. In the binary tree, search time will be O(log 8) = 3. We can see that 3 < 8 here. So the binary tree would be a better choice.

## Hypothesis

1- for reasonably well balanced binary search trees the average search time (to discover whether or not a particular element is present in thhe set) is O(log N) to the base 2 where N is the number of element in the set(the size of the tree).

2- A binary search tree requires approximately 1.39(log(n)) comparisons if keys are inserted in random order[4].

## Searching Technique

Searching a binary tree for a specific value can be a recursive or iterative process. This paper covers a recursive method (A recursive object is one that is defined partially in terms of itself)[5].

By examining the root node four cases arise. If the tree is null, the value we are searching for does not exist in the tree. Otherwise, if the value equals the root, the search is successful. If the value is less than the root, search the left subtree. Similarly, if it is greater than the root, search the right subtree. This process is repeated until the value is found or the indicated subtree is null. If the searched value is not found before a null subtree is reached, then the item must not be present in the tree [6].

The shape of the binary search tree clearly determines the efficiency of the binary search algorithm. The optimal binary tree for a given set of entries has minimal height (The height of the Binary Search Tree equals the number of links from the root node to the deepest node) and is well balanced in the sense that the root of every subtree of binary tree has as many left descendants as it has right descendants. In practice when there are dynamic insertions and deletions we are unlikely to maintain a perfectly balanced search tree [5].

## Proposed Design and Data Structure for Binary Search Tree

A range of randomly generated sets of suitably different sizes each of which is tested for the presence of a range of randomly generated integers. A set is represented by a pointer, named P therefore; the sets variable is P which points to a record consist of three fields namely:

Left: a pointer to a set of records.
Right: a pointer to a set of records.
Value: the value that the user is interested in.

In this section we shall work out the algorithms based on the best strategy to solve the problems, and compute the time complexity of the algorithms. This problem is related to searching recursively. Figure No.(1) shows the flowchart of the search technique.
A package (i.e. collections of functions and procedures) is developed; this package when declared at the outermost level of a program together with appropriate type declarations would make the following standard set operations available within the program:
**Create:** create an empty set (function).
**Insert:** Insert a given element into a given set (procedure), (do nothing if the element is already in the set).
**Comsion**: detect if a given element is present in a given set (function).
**Size**: return the size of a given set (function).
**Writeset**: write out the contents of a given set in ascending numerical order (procedure). The package implements the sets using binary search trees.

Once the binary tree is created, its elements can be retrieved in-order by recursively traversing the left subtree of the root node, accessing the node itself, then recursively traversing the right subtree of the node, continuing this pattern with each node in the tree as its recursively accessed. The package obtains the store required from the sets from the Pascal heap (i.e. the free store system), the domain of the sets provided is the positive integers available in Pascal, a pseudo-random number sequence is set up by declaring functions **Random and Randint.** And declaring and initializing the sequence carrier C. A comparison is made each time the value at a node of the binary search tree is inspected, Figure No.(2) shows part of the program code.

## Statistical Method

The chi-square method is used here to test the 'Goodness of fit' of a function. The value of chi-square is calculated by the following formula [7]:

Chi-sq = summation (sqr (Oi-Ei)/Ei)    Where
Oi = Obtained value in the ith observation;
Ei = Expected value in the ith observation

Our tested hypothesis is: for reasonably well balanced binary search trees the average search time is O (log N) to the base 2.
That means If it is balanced, worst casew is O( ln( N ) / ln( 2 ) ).Where N is the number of values and the ln(2) comes from a base 2 logarithm. If it is an unbalanced binary tree, O( ln( n ) / ln( 2 ) ) is best-case. Worst case is O (n).

The mathematical expression O( ln( N ) / ln( 2 ) ) is used to compute the Expected search time according to the size of the tree. This value is compared with the Obtained time (average number of comparisons).

The result after running the program is listed in Table No.(1) below.
The curve of Obtained time (time obtained from the experiment) compared to the expected time is shown in Figure No. (3).

## Conclusion

The first hypothesis, the conclusion is that binary search is said to run in a number of steps proportional to the logarithm of the length of the list being searched, or in O(log(N)),not exceeding this value if the tree is balanced and this value could be as average if it is unbalanced.
The value of Chi-square for sample of size 20 (D.F. 19) is (30.1), with 5% level of confidence. It means that if our Chi-square value is found to be greater than (30.1), we shall reject the above hypothesis. By doing so, we have a probability of 5% of making a wrong rejection.

Calculating our Chi-square value yield the value (9.5), therefore we shall not reject the above hypothesis so we accept it to be true.

Table No. (1) below shows the test is quite good where the size of the tree is 5,10,15,30,65, this leads us to assume that these trees are well balanced or almost balanced. In this experiment we have not ensure that all the trees are well balanced.

However, since 20 trees have been taken (which is not too small sample) and those inserted elements were generated from a pseudo random number, there is reason to believe that the trees are not too unbalanced, and the above hypothesis can be tested by the experimental result.

For the seconed hypothesis, a binary search tree requires approximately 1.39(log (n)) comparisons if keys are inserted in random order. Therefore, the average number of comparisons is only about 40% higher than the best possibl (See Table No.(1) the column **Obtained/Expected** which shows this fact), Figure No.(3) shows the comparisons between the obtained and the expected time of the binary search tree. However there was only one odd value and this is where the size of the tree is 85 which resulted in 56% higher and this is due to imbalance of the tree.

## References

1. Scott Mitchell and update January(2005) 'An Extensive Examination of Data Structure Using C# 2.0', Visual Studio 2005,4GuyFromRolla.com.

2. AAaron M Tenenbaum, Moshe J. Augenstein, Data Structures Using Pascal, 438,439, by Prentice-Hall, Inc.Englewood Cliffs,N.J.07632.

3. big O, notation, lastmodified on 1 September (2011). From Wikipedia,the free encyclopedia.

4. Suri Pushpa1, Prasad Vinod2, August (2007).Binary Search Tree Balancing Methods: A Critical StudyIJCSNS International Journal of Computer Science and Network Security,7:(8).

5. Derek Coleman, (1981) 'A Structured programming Approach to Data, the MacMillan Press LTD, London and Basingstoke Companies and representatives (p124, p126, p173).

6. ( Gilberg, R.; Forouzan, B. (2001), "8", *Data Structures: A Pseudocode Approach With C++*, Pacific Grove, CA: Brooks/Cole, 339,
(From Wikipedia, the free encyclopedia)

7. PPA 696 RESEARCH METHODS TESTS FOR SIGNIFICANCE, www.csulb.edu/msaintg.htm.

**Table No.(1): Obtained time (obtained from the proposed method) Compared with the Expected Time, the Ratio between the Two Shows that the Obtained Time is about 40% Higher as a Maximum**

| Tree size | Obtained time | Expected time | Obtained/Expected |
|---|---|---|---|
| 5 | 2.10E+00 | 2.32E+00 | 9.04E-01 |
| 10 | 3.60E+00 | 3.32E+00 | 1.08E+00 |
| 15 | 4.20E+00 | 3.91E+00 | 1.08E+00 |
| 20 | 5.00E+00 | 4.32E+00 | 1.16E+00 |
| 25 | 6.70E+00 | 4.64E+00 | 1.44E+00 |
| 30 | 5.00E+00 | 4.91E+00 | 1.02E+00 |
| 35 | 6.10E+00 | 5.13E+00 | 1.19E+00 |
| 40 | 6.20E+00 | 5.32E+00 | 1.16E+00 |
| 45 | 7.70E+00 | 5.49E+00 | 1.40E+00 |
| 50 | 7.30E+00 | 5.64E+00 | 1.29E+00 |
| 55 | 7.30E+00 | 5.78E+00 | 1.26E+00 |
| 60 | 8.00E+00 | 5.91E+00 | 1.35E+00 |
| 65 | 6.10E+00 | 6.02E+00 | 1.01E+00 |
| 70 | 7.80E+00 | 6.13E+00 | 1.27E+00 |
| 75 | 8.20E+00 | 6.23E+00 | 1.32E+00 |
| 80 | 9.00E+00 | 6.32E+00 | 1.42E+00 |
| 85 | 1.00E+01 | 6.41E+00 | 1.56E+00 |
| 90 | 8.00E+00 | 6.49E+00 | 1.23E+00 |
| 95 | 7.60E+00 | 6.57E+00 | 1.16E+00 |
| 100 | 9.00E+00 | 6.64E+00 | 1.35E+00 |

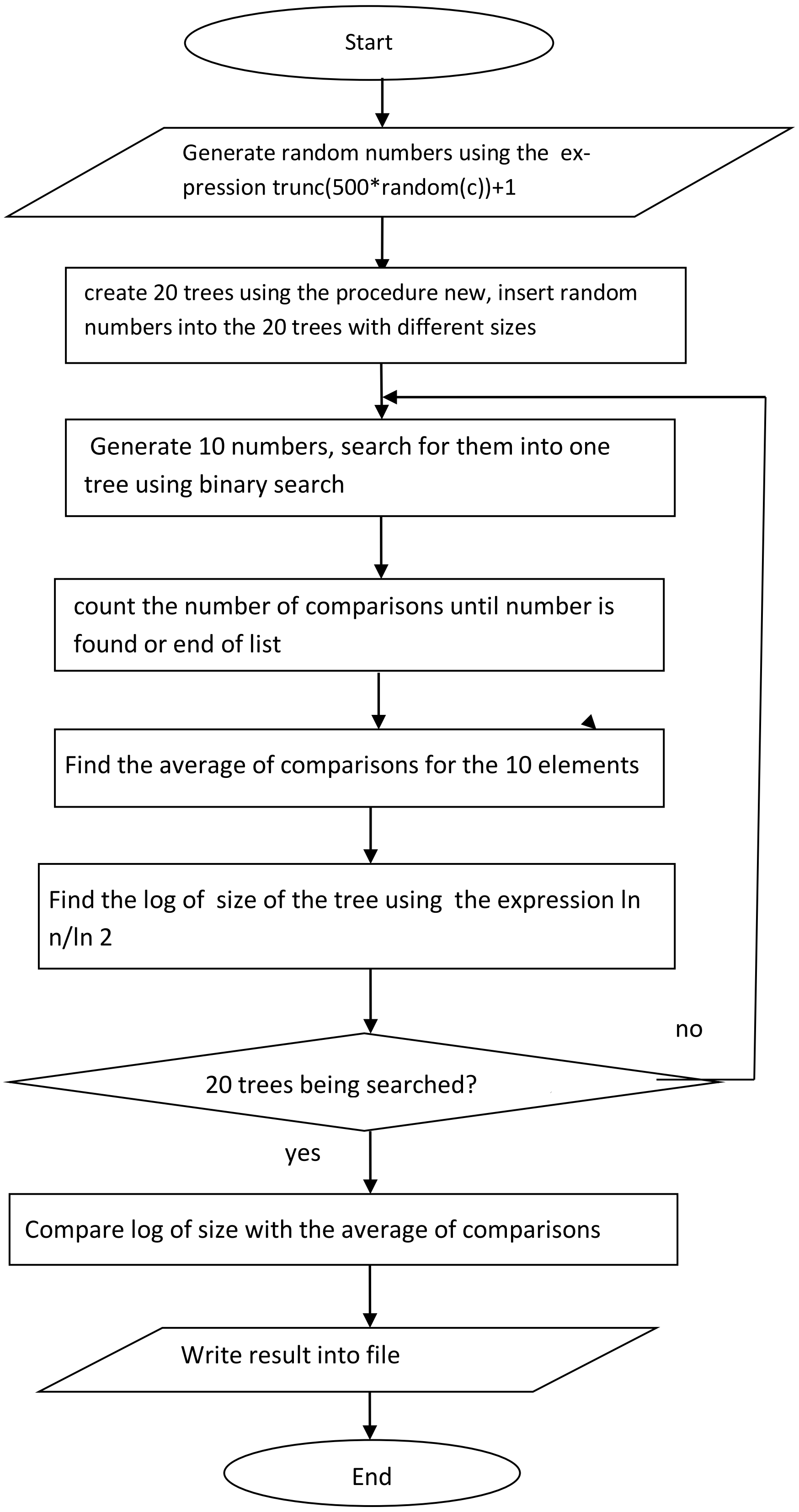
Start
Generate random numbers using the expression trunc(500*random(c))+1
create 20 trees using the procedure new, insert random numbers into the 20 trees with different sizes
Generate 10 numbers, search for them into one tree using binary search
count the number of comparisons until number is found or end of list
Find the average of comparisons for the 10 elements
Find the log of size of the tree using the expression ln n/ln 2
20 trees being searched?
no
yes
Compare log of size with the average of comparisons
Write result into file
End

**Figure No.(1):Flowchart of the Proposed Binary Search Technique**

program exp;

```
type p=^node;  node=record  left:p;  right:p;  value:integer;  end;

var  somesets:array[1..20] of p;  count,c,i,j:integer;  totcount,xx,ff:integer;

f:text; ch:char;  average,loog,cott:real;

function random(var c:integer):real;

begin  c:=26317*c+1;  random:=abs(c/32767)   end;

function randint:integer;

begin  randint:=trunc(500*random(c))+1   end;

function create:p;

begin  create:=nil;  end;

procedure insert(n:integer;var t:p);

var u:p;

begin  if t=nil then

begin  new(u);   u^.left:=nil;   u^.right:=nil;  u^.value:=n;  t:=u;  end;

if n<t^.value then insert(n,t^.left);  if n>t^.value then insert(n,t^.right);  end;

function comsion(n:integer;t:p):integer;

var found:boolean;

begin

found:=false;count:=0;  while(not found) and (t<>nil) do

begin  if t^.value=n then

begin  found:=true;  count:=count+1;  end

else  if t^.value>n then

begin  t:=t^.left;  count:=count+1;  end

else  begin  t:=t^.right;  count:=count+1  end;  end;  comsion:=count;  end;
```

**Figure No. (2): Part of Binary Search Program Code**

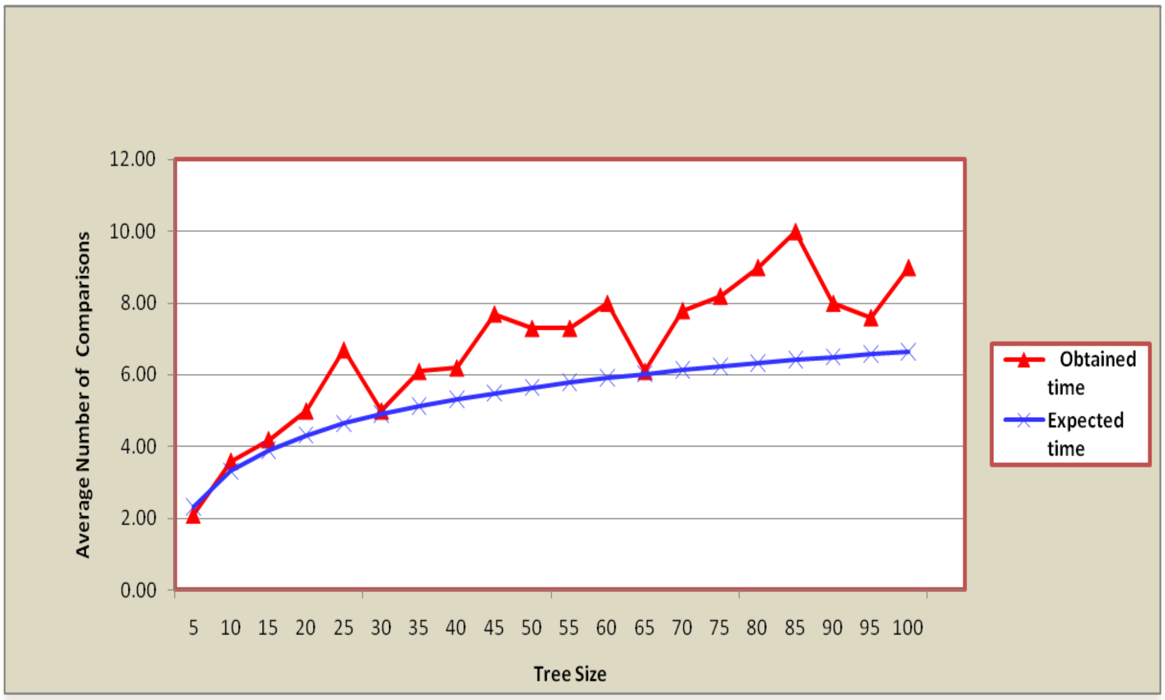

**Figure No.(3) Comparisons between the Obtained and the Expected Time of the Binary Search Tree Technique**

# تجربه محاكاة لبرهنة الافتراض النظري لتعقيدات الوقت للبحث الثنائي الشجري

**منى مهدي صالح**
الكليه التقنيه الادارية/ هيئه التعليم التقني/ بغداد
munaouceal@yahoo.co.uk

**استلم البحث في 28 نيسان 2013، قبل البحث في 4 كانون الاول 2013**

## الخلاصه

لطالما تم التاكيد ان تطبيق تقنية البحث الثنائى الشجرى لمعالجة مجموعة ما له افضلية على تطبيق تقنية القائمة الموصولة، اذ انه فى البحث الشجرى الثنائى المتوازن فان معدل وقت البحث (للتحري عن وجود عنصر معين فى المجموعة) هو O(log N) للاساس 2 اذ N هو عدد العناصر فى الشجرة او حجم الشجرة.
هذا البحث يقدم تجربة لقياس ومقارنة وقت البحث الثنائى الشجرى المستحصل مع الوقت المتوقع(نظريا) ،هذه التجربة العملية اثبتت صحه الفرضية، نفذت هذه التجربه باستخدام برنامج تربو باسكال مع تطبيق تقنية الاستدعاء الذاتي،واستخدام طريقة احصائية لاثبات الفرضية اعلاه، تم تقدير وقت البحث بوساطة عدد المقارنات المطلوبة.

**الكلمات المفتاحية:** Time Complexity, Binary Search Tree, Big O Notation, Recursive Method, Chi-Square.