Homework 7 - Binary Search Trees

Problem 1:

What does it mean if a binary search tree is a *balanced tree*?

ANS

It means that in this binary tree the depth of the two subtrees of every node never differ by more than one. The binary search tree is height-balanced.

Problem 2:

What is the big-Oh search time for a balanced binary tree? Give a logical argument for your response. You may assume that the binary tree is perfectly balanced and full.

ANS

The big-Oh search time for a balanced binary tree is $O(logn)$.It is similar with the binary search.

Assuming the number of nodes of a balanced binary tree is n, the height of tree is h. The binary tree is perfectly balanced and full, we could get that $n = 2^h - 1$, which also means $h = log(n+1)$. *T(n)* is the average time of we search one target value in a balanced tree.

When $h = 2, n = 3, T(n) = (1*1 + 2*2)/3$

When $h = 3, n = 7\ T(n) = (1*1 + 2*2 + 3*4)/7$

When $\boldsymbol{h = h, n = n,\ T(n) = (1*1 + 2*2 + 3*4 + ... + h*2^{(h-1)})}$/n ①

Then we could get,

$1*1 + 2*2 + 3*4 + ... + h*2^{(h-1)}$ = $1*2^0 + 2*2^1 + 3*2^2 + ... + h*2^{(h-1)}$ **= SUM** ②

$2*\mathrm{SUM} = 2*(1*2^0 + 2*2^1 + 3*2^2 + ... + h*2^{(h-1)})$

=$1*2^1 + 2*2^2 + 3*2^3 + ... + (h-1)*2^{(h-1)} + h*2^h$ ③

Then, we use ③ − ②

SUM = $h*2^h - (2^0 + 2^1 + 2^2 + 2^3 + ... + 2^{(h-1)})$

$\boldsymbol{A} = 2^0 + 2^1 + 2^2 + 2^3 + ... + 2^{(h-1)}$ is the sum of geometric sequence.

Then, $\mathbf{A = 2^h - 1},\ \boldsymbol{h = log(n+1)}$

SUM = $h*2^h - (2^h - 1) = (h-1)*2^h + 1 = [log(n+1) - 1]*2^{log(n+1)} + 1$

$= [log(n+1) - 1]*(n+1) + 1$

$= log(n+1)*(n+1) - n$

So, $\boldsymbol{T(n) = SUM/n = \frac{n+1}{n} log(n+1) - 1}$

The big-Oh search time for a balanced binary tree is $\boldsymbol{O(logn)}$.

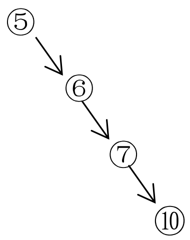
Problem 3:

Now think about a binary search tree in general, and that it is basically a linked list with up to two paths from each node. Could a binary tree ever exhibit an O(n) worst-case search time? Explain why or why not. It may be helpful to think of an example of operations that could exhibit worst-case behavior if you believe it is so.

ANS

The big-Oh search time for a binary tree could be *O(n).*

The worst-case is the binary tree is totally unbalanced, it only has a left path or a right path, which means that we can think of the tree as a linked list. The big-Oh search time is same as a linked list.



In the worst case, we need to traverse all nodes of the unbalanced tree, find the target value or not.

Problem 4:

What is the recurrence relation for a binary search? Your answer should be in the form of $T(n) = aT(n/b) + f(n)$. Clearly state the values for *a*, *b* and *f(n)*.

ANS

Pseudocode:

BinarySearch(array A[], int value, int first, int last)

```
1   if( first > last)
2       return -1// not found target value
3   int mid = ( first + last) / 2;
4   if ( value == A[mid]) then
5       return mid;
6   else
7       if( value  <  A[mid])
8           BinarySearch(A, value, first, mid - 1);
9       else if( value  >  A[mid])
10          BinarySearch(A, value, mid + 1, last);
```

Assuming *T(n)* is the time we spend when we use binary search and the number of array is n.

When n = 1, the algorithm executes from line1 to line5, It takes a "constant" time, T(1) = 1.

When n > 1, the algorithm will execute on line8 or line 10. For both line, they run with the subarray which has $\frac{n}{2}$ elements. The sum of running time from line6 to line 10 is always $\mathrm{T}(\frac{n}{2})$.

Then we have,

$$\begin{cases} T(1) = 1, & n = 1 \\ T(n) = T(\frac{n}{2}) + T(1), & n > 1 \end{cases}$$

So, $T(n) = aT(n/b) + f(n)$ = $T(\frac{n}{2}) + T(1), n > 1$. $\mathbf{a = 1, b = 2\ f(n) = 1}$

Problem 5:

Solve the recurrence for binary search algorithm using the *substitution method*. For full credit, show your work.

ANS

According to #4, we have

$$\begin{cases} T(1) = 1, & n = 1 \\ T(n) = T(\frac{n}{2}) + T(1), & n > 1 \end{cases}$$

$T(1) = 1,$

$T(2) = 1 + 1 = log_2 2 + 1$

$T(4) = 2 + 1 = log_2 4 + 1$

$T(8) = 3 + 1 = log_2 8 + 1$

$T(16) = 4 + 1 = log_2 16 + 1$

We can deduce that $T(n) = logn + 1$, the big-Oh is $O(logn)$.

Problem 6:

Confirm that your solution to #5 is correct by solving the recurrence for binary search using the *master theorem*. For full credit, clearly define the values of *a*, *b*, and *d*.

ANS

The definition of the master theorem is:

If $T\ (n)\ = a * T(\frac{n}{b}) + n^d$ *for a>1, b>1, d≥0*

There are three cases:

if $d > log_b a$ $\quad T(n) = O(n^d)$ ①

else if $d = log_b a$ $\quad T(n) = O(n^d logn)$ ②

else if $d < log_b a$ $\quad T(n) = O(n^{log_b a})$ ③

According to #5, we have

$$\begin{cases} T(1) = 1, & n = 1 \\ T(n) = T(\frac{n}{2}) + T(1), & n > 1 \end{cases}$$

when n > 1, we can know that *a =1*, *b* = 2*, d = 0*

According to the Master Theorem, it's the second case,

d = $log_b a = 0,\ T(n) = O(n^d logn) = O(logn)$

The big-Oh is the same as the complexity of binary search algorithm with recursion, thus it's definitely correct.