Department of Electrical and Computer Engineering
McGill University

ECSE-322 Computer Engineering

24 January 2008

Problem Set 3 - Solutions

1. Give upper and lower bounds for the average level of a node in:
   (a) a complete binary tree of height N
   (b) a binary tree of height N.

Solution:

(a) The average level of the nodes in a complete binary tree can vary even though the tree height is fixed, because the last level (level N) can contain anywhere between 1 and 2^{N-1} nodes.

The average level can be computed by a weighted sum:
avg = (1 * #nodes at level 1 + 2 * # nodes at level 2 + … + N * # nodes at level N) /
(# nodes at level 1 + # nodes at level 2 + … + # nodes at level N)

If level K of the tree is completely occupied (has the maximum possible number of nodes), then the number of nodes at level K is 2^{K-1}.

The highest average level for a complete binary tree (the upper bound) is obtained when the last level of the tree is fully occupied.

high avg = $(1 * 2^0 + 2 * 2^1 + \ldots + (N-1)*2^{N-2} + N*2^{N-1}) / (2^0 + 2^1 + \ldots + 2^{N-2} + 2^{N-1})$

Since $2^K = 2^{K+1} – 2^K$, the sum $2^0 + 2^1 + \ldots + 2^{N-2} + 2^{N-1}$ can be rewritten as
$(2^N - 2^{N-1}) + (2^{N-1} - 2^{N-2}) + \ldots (2^1 – 2^0) = 2^N – 1$, after cancellations.

The sum $1 * 2^0 + 2 * 2^1 + \ldots + (N-1)*2^{N-2} + N*2^{N-1}$ can be rewritten by expanding each term as follows:

$$
\begin{array}{ll}
 & 2^0 + 2^1 + \ldots + 2^{N-2} + 2^{N-1} \\
+ & \quad 2^1 + \ldots + 2^{N-2} + 2^{N-1} \\
+ \ldots & \\
+ & \quad\quad\quad\quad\quad 2^{N-1}
\end{array}
\Bigg\} \text{ N rows}
$$

$= (2^N - 2^0) + (2^N - 2^1) + \ldots + (2^N - 2^{N-1})$ (by adding up the elements of each row)

$= N*2^N - (2^0 + 2^1 + \ldots + 2^{N-1}) = N*2^N - (2^N - 1) = (N-1) * 2^N + 1.$

We conclude that

high avg. $= ((N-1) * 2^N + 1) / (2^N - 1) = N-1 + (N/(2^N - 1))$.

The lowest average node level for a complete binary tree is obtained when the last level of the tree has just one node.

low avg $= (1 * 2^0 + 2 * 2^1 + \ldots + (N-1)*2^{N-2} + N*1) / (2^0 + 2^1 + \ldots + 2^{N-2} + 1)$

$= ((N-2) * 2^{N-1} + 1 + N) / ((2^{N-1} - 1) + 1)$ (using the sum calculations made above)

$= N-2 + (N+1)/ 2^{N-1}$

(b) The highest average node level is obtained when the tree is a complete binary tree with the last level full – same as in Part (a).

The lowest average level is obtained when the tree has just one node at each level.

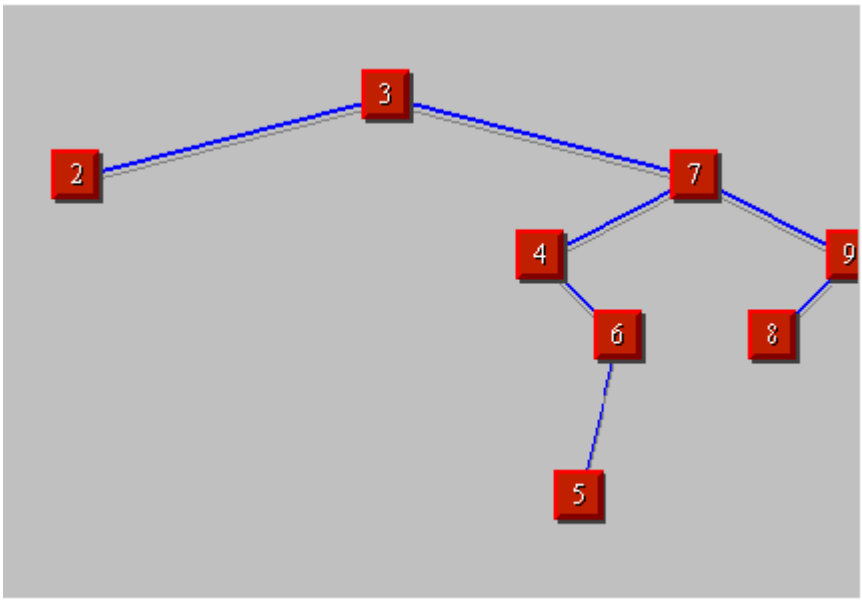
low avg. $= (1*1 + 2*1 + \ldots + N*1) / N = (N(N+1)/2) /N = (N+1) / 2$

Notice that all the bounds in Part (a) and (b) are O(N). Moreover, both the upper and lower bounds in Part (a) are close to N, which is consistent with the fact that the majority of the nodes in a complete binary tree are on the last couple of levels.

2. Show the result of inserting 3, 7, 4, 6, 9, 3, 5, 8, and 2 into an initially empty binary search tree. Next, extract the items using inorder, preorder and postorder traversal. For the resulting tree,

(a) What can you tell about the extracted order of each method?
(b) Is this a full tree?
(c) Identify the root
(d) Identify the leaves
(e) What are the degree and level of 9?
(f) What is the height of the tree?

Solution:



Inorder traversal:
2,3,4,5,6,7,8,9

Preorder traversal:
3,2,7,4,6,5,9,8

Postorder traversal:
2,5,6,4,8,9,7,3

(a) Inorder is the one that extracts it in the correct order.
(b) Not a full tree.
(c) Root: 3
(d) Leaves: 2,5,8
(e) Degree 1, level 3
(f) Height 5

Please note that the duplication of the element 3 is handled by the algorithm in a fashion that loses the element (i.e. it is absorbed in the previous element 3 node). Other algorithms may permit duplication by permitting <= or >= operations for insertions in the left and right directions. This is an implementation issue.

3. Design and implement a program for accepting data as it arrives and placing it in a binary tree structure. The data is in the form of integers. The rule to be followed in constructing the tree is that anything smaller than the root gets placed to the left of the root, anything larger is placed to the right.

(a) The design should include a flowchart and description.
(b) Implement your design as a program.

Solution:

This has been covered in the lectures in class and in the overheads. The design is described in the overheads.

4. In placing numbers in a binary tree, the first number received is to be used as the root. If the total number of data items received is n,

(a) discuss the effect on the performance of the tree storage algorithm (i.e. the algorithm designed in question (3) if the majority of the numbers (say 80%) are less than the root value.
(b) discuss the effect on the performance of the tree storage algorithm if the data is equally distributed around the root value.
(c) If the data is of the form given in part (a), how would you rearrange the tree (once data has been stored in it) to achieve the structure which results in part (b)?

Solution:

In general, the tree storage algorithm operates as O(NlogN) since, for a complete binary tree, the number of levels is logN and we have N items to store. However, if 80% of the values are to be stored on the left, the tree will not be complete and the height on the left will be more than log N. If the left hand side consists of a complete sub-tree, then the number of levels will be log(0.8N) rather than logN – 1. Thus the storage performance will be worse.

If the data is equally distributed around the root node, the performance should be O(NlogN).

The answer to the third part requires the development of an algorithm for height balancing the tree, i.e. restructuring it to provide logN levels. One method of doing this, which is definitely not optimal, is to read the data from the tree using an inorder traversal, choose the center of the list and use this as the root node of a new tree. The new tree can be constructed from the old one by moving the entry point to the tree to the center value node. Then, the link to the new root from its previous parent is reversed, i.e. the original parent becomes a child of the new root and the appropriate old sub-tree of the new root (left or right) is linked to the previous parent as a new sub-tree (right or left).

Note that the average case time for tree storage of N items is O(N log N) – note relationship between the number of levels in the resulting tree and the number of iterations of Quicksort, as discussed at the lectures.

The performance of tree storage has interesting consequences. If N data items are stored in O(NlogN) time on average, each data item takes on average O(log N) time. Also, since the storage tree is a binary tree, it supports binary search; hence, the search for an item of a given value takes O(log N) time on average. This gives us a data structure where *both* the insert and search operations have O(log N) performance!

5. Compute the time complexity of the operations: insertion and deletion (including search) for each of the following implementations of an ordered list of size N:

   (a) Sorted array
   (b) Sorted linked list
   (c) Binary tree

Solution:

Implementation	Insertion (including search)	Deletion (including search)
Sorted array	O(log N) search + O(N) moves	O(log N) search + O(N) moves
Sorted linked list	O(N)	O(N)
Binary search tree	O(log N)	O(log N)

Which is the most efficient?

Solution:

The binary search tree is the most efficient implementation of an ordered list.