CS311: Homework #4

Due on October 11, 2013 at 4:30pm

Professor Lathrop Section 3

Josh Davis

Problem 1

Consider the binary search tree code given and prove the two propositions.

Solution

Part One Prove that the best case running time of **BUILD-BST** is $\Omega(n \lg n)$.

Proof. To prove that the best case running time of **BUILD-BST** is $\Omega(n \lg n)$, we will define a recurrence for **BUILD-BST**, $T(n)$, and then show that $T(n) \in \Omega(n \lg n)$ by showing that $0 \leq c \cdot n \lg n \leq T(n)$ for some constant $c > 0$ and all $n \geq n_0$.

To prove the worst case, we need to understand how **BST.add()** works. It recurses into either the left or right subtree dependent on the values of each node.

To illicit best case run-time, it follows that during each recursion, we cut down the size of our problem by a factor of 2. This would give the recurrence as follows: $T(n) = T(n/2) + \Theta(1)$ where the work done at each step is constant. We can now use the master method.

By applying the master method, we get $a = 1$, $b = 2$ and $f(n) = \Theta(1)$. Which gives $n^{\log_b a} = n^{\log_2 1} = n^0 = 1$. Since $f(n) \in \Theta(1)$, this puts us into the second case where $T(n) = \Theta(n \lg n)$.

By using the following knowledge:

$$\exists c_1 \exists c_2 \forall n \geq n_0,\ c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

It follows that when a function $T(n) \in \Theta(g(n))$, it also is $T(n) \in \Omega(g(n))$ by the definition of $\Theta(n)$. Thus we have proven the best case running time of **BUILD-BST** is $\Omega(n \lg n)$. □

Part Two Prove that the worst case running time of **BUILD-BST** is $O(n^2)$.

Proof. To prove that the worst case running time of **BUILD-BST** is $O(n^2)$, we will define a recurrence for **BUILD-BST**, $T(n)$, and then show that $T(n) \in O(n^2)$ by showing that $T(n) \leq c \cdot n^2$ for some constant $c > 0$ and all $n \geq n_0$.

To prove the worst case, we need to understand how **BST.add()** works. It recurses into either the left or right subtree dependent on the values of each node.

To illicit worst case run-time, it follows that during each recursion, we only cut down the size of our problem by 1. This would give the recurrence as follows: $T(n) = T(n-1) + \Theta(1)$ where the work done at each step is constant. By using the substitution method we get the following:

$$\begin{aligned} T(n) &= T(n-1) + \Theta(1) \\ &\leq (c \cdot (n-1))^2 + \Theta(1) \\ &= c^2 \cdot (n^2 - 2n + 1) + \Theta(1) \\ &\leq c \cdot n^2 - c \cdot 2n + c + \Theta(1) \\ &\leq c \cdot n^2 \end{aligned}$$

By using the substitution method we can see that the worst case run time of **BUILD-BST** is $O(n^2)$ and thus our proof is complete. □

Problem 2

Consider an AVL tree where every node in the tree has a balance factor of 0. Prove using induction that any such AVL tree is also full: the d^{th} level of the tree (where the root is in level 0) has either 2^d or 0 nodes.

Solution

Proof. Prove that an AVL tree with a balance factor of 0 for every node is full. A full binary tree has either 2^d or 0 nodes at the d^{th} level.

Base
Consider a tree where with a single node, the root, and where it has a balance factor of 0. As given, we consider this root node to be at depth $d = 0$. This gives us $2^d = 2^0 = 1$ node. Since this is perfectly balanced and also consists of just 1 node. Adding any nodes would offset the balance factor and would result in being more than 1 node. Thus the base case holds.

Step
We wish to show for a AVL tree with a depth of d, if we expand it so that the depth equals $d + 1$) and the balance factor is still 0, it must create a "full" binary tree in that at the last depth, $d + 1$ there will still be 2^{d+1} nodes.

Consider two AVL trees T_1 and T_2 both with a depth d and balance factors of 0. If we were to construct a new AVL tree T by taking a new node and adding T_1 and T_2 as subtrees to the new root, we would end up with a new tree. The new tree would then have a depth of $d + 1$ because the new root adds one to the depth. Since both T_1 and T_2 both have the same number of nodes, our new tree has a balance factor of 0.

Since we are taking two AVL trees with balance factors of 0, we get the following at the last level:

$$\text{nodes at level } d + 1 = \text{nodes at level d of } T_1 + \text{nodes at level d of } T_2$$

By assuming the induction hypothesis we get the following:

$$\begin{aligned}\text{nodes at level } d + 1 &= 2^d + 2^d \\ &= 2 \cdot 2^d \\ &= 2^{d+1}\end{aligned}$$

Thus we can see that adding any more nodes would unbalance the tree and it wouldn't be full anymore. Thus the proof is complete. □

Problem 3

Write pseudocode that when given a list of elements sorted in ascending order, return a complete BST that contains precisely the elements in the input list. The algorithm should run in linear time.

Solution
The algorithm works by taking the median of the list and then setting it as the root element in the BST then recurses until the range consists of only one element. The pseudocode of the algorithm is below:

```
function BuildBST($list, start, end$)
    if $start > end$ then
        return NULL
    else if $start = end$ then
        return new Node($list[start]$)
    end if

    $mid \leftarrow start + (end - start)/2$
    $root \leftarrow$ new Node($list[mid]$)

    $root.left \leftarrow$ BuildBST($list, start, mid - 1$)
    $root.right \leftarrow$ BuildBST($list, mid + 1, end$)

    return $root$
end function
```

Problem 4

Let H be an empty hash table with $n = 6$ bins. Assume that H stores integers according to the hash function $h(x) = x$ and the probing function:

$$P(x,i) = \begin{cases} (P(x, i-1) + i) \bmod n & i > 0 \\ h(x) \bmod n & i = 0 \end{cases}$$

1. Show the configuration of H after the integers 6, 1, 3, 10, and 4 are added.
2. What happens when the table is checked for the inclusion of the integer 0? Explain your answer.

Solution

Part One
The locations as follows:

1. When 6 is to be added: the place is $P(6,0) = 6 \bmod 6 = 0$ which is empty and is placed there. $H = [6, null, null, null, null, null]$
2. When 1 is to be added: the place is $P(1,0) = 1 \bmod 6 = 1$ which is empty and is placed there. $H = [6, 1, null, null, null, null]$
3. When 3 is to be added: the place is $P(3,0) = 3 \bmod 6 = 3$ which is empty and is placed there. $H = [6, 1, null, 3, null, null]$
4. When 10 is to be added: the place is $P(10,0) = 10 \bmod 6 = 4$ which is empty and is placed there. $H = [6, 1, null, 3, 10, null]$
5. When 4 is to be added: the place is $P(4,0) = 4 \bmod 6 = 4$ which isn't empty. Therefore we probe again: $P(4,1) = (P(4,0) + 1) \bmod 6 = (4 + 1) \bmod 6 = 5$ which is empty and it is placed there. $H = [6, 1, null, 3, 10, 4]$.

Part Two
When the table is checked for the inclusion of 0, we need to look calculate the hash of where 0 should be placed. We then check to see if 0 is in that place. If nothing has been placed in that location of the hash table, we know that 0 cannot be anywhere. However, if a value is there that isn't 0, we need to run the probing until we run into a value that doesn't exist.

The sequence of probes and checking will look like the following:

1. First we check $P(0,0) = 0 \bmod 6 = 0$ to see if it is empty. It isn't *null* and $0 \neq 6$. Therefore we continue.
2. Second we check $P(0,1) = (P(0,0) + 1) \bmod 6 = (0 + 1) \bmod 6 = 1$ to see if it is empty. It isn't *null* and $0 \neq 1$. Therefore we continue.
3. Third we check $P(0,2) = (P(0,1) + 1) \bmod 6 = (1 + 1) \bmod 6 = 2$ to see if it is empty. It is *null* which ends the probing sequence. Therefore the check would return $false$ because 0 cannot be in the hash table under open addressing.

Problem 5

Consider the algorithm given to traverse a rooted binary tree where **VISIT** is a constant-time function. Using the master theorem, give a tight bound on the running time of **TRAVERSE** as a function of the number of nodes in t.

Solution
First we must come up with a recurrence for the given algorithm. The recurrence for **TRAVERSE** is: $T(n) = T(n/2) + T(n/2) + \Theta(1) + 1$. This is because each recursive call divides the binary tree into two and conquers each branch recursively.

Since **visit** is a constant-time function, it can be represented in the recurrence as $\Theta(1)$. The extra addition of 1 is to capture the if statement check. Similarly, this check has a constant run time which can be absorbed into the $\Theta(1)$ but left for clarity sake.

By combining the recurrence we get $T(n) = 2 \cdot T(n/2) + \Theta(1)$. Using the master method, we get $a = 2$, $b = 2$, and $f(n) = \Theta(1)$. This gives us $n^{\log_b a} = n^{\log_2 2} = n^1 = 1$.

No matter what ϵ equals as long as $\epsilon > 0$, it will always grow faster than a constant function. Therefore, this puts us into the first case of the master method. This says that $T(n) = \Theta(n)$. Thus it is a tight bound on the running time of the **TRAVERSE** algorithm and our analysis is complete.

Problem 6

Consider the following recurrence relation:

$$T(n) = \begin{cases} 1 & n = 1 \\ 7 \cdot T(\frac{n}{2}) + 1 & n > 1 \end{cases}$$

1. Use the master theorem to show that $T(n) \in \Theta(n^{\log_2 7})$.
2. Use induction to show that $T(n) = \frac{1}{6} \cdot (7 \cdot n^{\log_2 7} - 1)$. You may assume that n is a power of two.

Solution

Part One

We have $a = 7$, $b = 2$ and $f(n) = 1$. According to the Master Method, this gives us:

$$n^{\log_b a} = n^{\log_2 7} \in \Theta(n^{\log_2 7})$$

This puts us in case 1 of the master method because $f(n) = O(g(n))$ where $f(n) = 1$ and when $\epsilon \approx .8$, $g(n) = n^{\log_2 7 - \epsilon} = n^{\log_2 7 - \epsilon} = n^2$.

To prove that we are in case 1, we just need to find a c and n_0 such that for all $n > n_0$ the inequality $1 \leq c \cdot n^2$ holds. If we let $n_0 = 1$ and $c = 1$ the inequality holds, thus proving $f(n) = O(g(n))$.

Now that we've verified that we are in case 1 of the master method, we now know that $T(n) = \Theta(n^{\log_2 7})$.

Part Two

Proof. We will use induction to prove that $T(n) = \frac{1}{6} \cdot (7 \cdot n^{\log_2 7} - 1)$.

Base

Consider the base case when $n = 1$. It is trivial because $T(1) = 1$ which is equal to $\frac{1}{6}(7n^{log_2 7} - 1) = \frac{1}{6}(7 \cdot 1^{log_2 7} - 1) = \frac{1}{6}(7 - 1) = 1$.

Step

To prove the induction step, we can assume that $n > 1$ and that n is a power of two. The induction hypothesis is thus $T(n) = \frac{1}{6}(7n^{log_2 7} - 1)$. This give us:

$$\begin{aligned} T(n) &= 7 \cdot T(n/2) + 1 \\ &= 7(\frac{1}{6}(7(n/2)^{log_2 7} - 1)) + 1 \\ &= 7(\frac{1}{6}(n^{log_2 7} - 1)) + 1 \\ &= 7(\frac{1}{6}n^{log_2 7} - \frac{1}{6}) + 1 \\ &= \frac{7}{6}n^{log_2 7} - \frac{7}{6} + 1 \\ &= \frac{1}{6}(7n^{log_2 7} - 7 + 6) \\ &= \frac{1}{6}(7n^{log_2 7} - 1) \end{aligned}$$

□

Problem 7

Using the given operations of a priority queue do the following:

1. Provide a pseudocode implementation of the **enqueue**(e) and **dequeue**() operations of a queue that uses a priority queue to store its data.
2. Provide pseudocode implementations of the **push**(e) and **pop**() operations of a stack that uses a priority queue to store its data.

Solution
Higher priority means a lower number for this pseudocode.

Part One

```
 $index \leftarrow 0$
 $queue \leftarrow$ new PriorityQueue( )

function ENQUEUE($e$)
    $queue$.ENQUEUE($e, index$)
    $index \leftarrow index + 1$
end function

function DEQUEUE( )
    return $queue$.DEQUEUE( )
end function
```

Part Two

```
 $index \leftarrow INT\_MAX$
 $queue \leftarrow$ new PriorityQueue( )

function ENQUEUE($e$)
    $queue$.ENQUEUE($e, index$)
    $index \leftarrow index - 1$
end function

function DEQUEUE( )
    return $queue$.DEQUEUE( )
end function
```

Problem 8

Consider the algorithm given for quicksort. Using the knowledge that quicksort depends on the behavior of **SELECT-PIVOT**, assume that **SELECT-PIVOT** runs in constant time and **PARTITION** runs in time $\Omega(end - start)$, do the following:

1. What **SELECT-PIVOT** behavior will lead to worst case **QUICKSORT** behavior? Use the master theorem to derive a tight bound on **QUICKSORT**'s worst case running time.
2. What **SELECT-PIVOT** behavior will lead to the best case **QUICKSORT** behavior? Use the master theorem to derive a tight bound on **QUICKSORT**'s best case running time.

Solution

Part One
To derive a tight bound on the worst case running time, we need to come up with a recurrence that captures the worst case running time of the algorithm.

Since **QUICKSORT** is broken up into two subproblems, from *start* to $mid - 1$ and then *mid* to *end*, the worst thing that could happen is if the mid selected is equal to the end and only cuts down the list by a single element each time. This would cause the recurrence to be the following:

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(end - start) \\ &\leq T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) \end{aligned}$$

We can't use the master method in this case but we can either use the substitution method which will give us an upper bound of $\Theta(n^2)$. Intuitively, this makes sense because for every recursive call from n to 0, we do $\Theta(n)$ work which gives us a quadratic run time.

Part Two
To derive a tight bound on the best case running time, we will come up with a recurrence again.

In the best case, the divide and conquer will operate the best when each problem is as close to the same size as possible. This means that it will half the problem size for each recursive call. This gives us the recurrence of the following:

$$\begin{aligned} T(n) &= T(n/2) + T(n/2) + \Theta(end - start) \\ &\leq T(n/2) + T(n/2) + \Theta(n) \\ &= 2 \cdot T(n/2) + \Theta(n) \end{aligned}$$

Using the master method, this gives us $a = 2$, $b = 2$ and $f(n) \in \Theta(n)$. Thus $n^{\log_b a} = n^{\log_2 2} = n^1 = n$. Since $f(n) \in \Theta(n)$, this puts us in the second case. The second case then gives us the following best case run time: $T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(n^{\log_2 a} \lg n) = \Theta(n \lg n)$.

Since we've found the worst case and best case running times, our solution is now complete.

Problem 9

A 10-foot rope is stretched across a gap. n ants are distributed on the rope and each ant faces either direction. At the same instant, the ants begin to walk, moving forward at a rate of 10 feet per minute. If two ants collide, they immediately reverse direction. When an ant reaches the end of the rope, it steps off the rope and can no longer block the other ants.

1. What is the largest possible amount of time that can elapse before the last ant steps off the rope?
2. In what situation does the worst-case time arise?

Solution

Part One
When two ants bounce off each other, it has no affect on the actual travel times. If the ants didn't bounce and just continued to travel, the solution would still be the same. Therefore since the rope is 10 feet long and the ants travel at 10 ft per minute, the longest time it would take for the rope to be clear of ants is 1 minute.

Part Two
The worst case time would arise whenever an ant is placed at the very edge of the rope. The ant that will be left on the rope last will always be the ant that is farthest from the respective end that it faces.

Therefore even if there are two ants placed on the edge of the rope facing inwards, they will both fall off at after 1 minute because they will meet in the middle and then reverse and travel the remaining half of the rope.