CS 263
Data Structures
ASSIGNMENT # 2
DUE DATE: Tuesday, October 22, 2013

If you are working in a group of 2 or three, please submit one copy with all of your names and student numbers on each sheet. Please use a fresh sheet of paper for each question.

1. Question 2 from Homework 1. (I gave you an extension on this question.)

   Solution: Assume that N the number of elements stored is equal to $2^n - 1$. Take an array where the top $\log n - 1$ levels have key $k = 2$ and then the last n^{th} level has one 2, followed by all 1's.

   For example: Take $N = 31 = 2^5 - 1$ (so $n = 5$). This is what the heap originally looks like:

   Consider what happens when we do the first 2^{n-1} DeleteMax moves. In our example, this is the first 16 DeleteMax moves. This will remove all elements with key 2 from the tree and since the tree always stays perfectly balanced, and what we are left with should be a balanced tree of height $n-1$ consisting of only keys with value 1. In our case, a height 4 tree consisting of ALL 1's.

   In particular, at this point in the algorithm, the last level, the $(n-1)^{st}$ level, is all 1's. But how did these 1's get there? These got there by first putting them at the root and then bubbling them all the way down to level $n-1$. So all of these elements at level $n-1$ should each require $(n-1)$ swaps in order to bubble them down from the root. In our example there are 8 of them and in general there are 2^{n-2} of them, and each of them requires $(n-1)$ swaps for a total of $2^{n-2}(n-1) = \Omega(n2^n) = \Omega(NlogN)$ steps.

2. Suppose 3 values A, B, and C are chosen uniformly and independently from the set of integers $\{1, \dots, r\}$, where $r \geq 1$.

   (a) What is the probability that all three values are the same? Briefly justify your answer.
   Solution:
   $\frac{1}{r^2} = \frac{1}{r} \times \frac{1}{r}$.
   Once the value for A has been chosen, the probability that B has the same value is $1/r$. The same is true for C. Since these are independent random variables, we can simply multiply the probabilities.

   Alternatively, there are r^3 triples of elements, each with the same probability. Of these, r triples have all three values the same. Thus the probability is $\frac{r}{r^3} = \frac{1}{r^2}$.

   (b) What is the probability that all three values are different? Briefly justify your answer.
   Solution:
   $\frac{(r-1)(r-2)}{r^2}$.
   Once the value for A has been chosen, the probability that B has a different value is $(r-1)/r$. Once different values for A and B have been chosen, the probability that C has a different value is $(r-2)/r$. Then $\Pr[A, B, C \text{ distinct }] = \Pr[A \neq B] \cdot \Pr[A, B, C \text{ distinct } | A \neq B] = \frac{r-1}{r} \cdot \frac{r-2}{r}$.
   Alternatively, of the r^3 triples of elements, there are r ways to choose A, $r-1$ ways to choose B different from A and $r-2$ ways to choose C different from A and B. Thus the probability is $\frac{r(r-1)(r-2)}{r^3} = \frac{(r-1)(r-2)}{r^2}$.

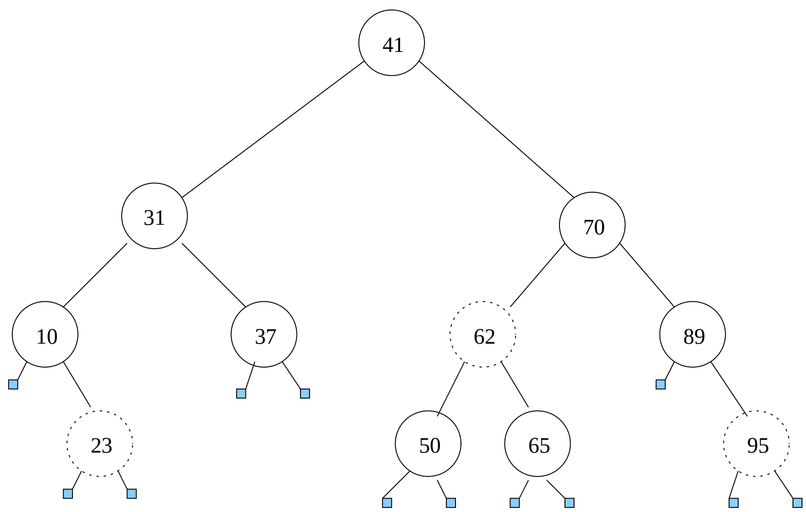
   (c) What is the expected number of different values? Briefly justify your answer.
   Solution:

The probability that there are two different values is $1 - \frac{1}{r^2} - \frac{(r-1)(r-2)}{r^2} = \frac{3(r-1)}{r^2}$, since this is the only other possibility.

Thus the expected number of different values is

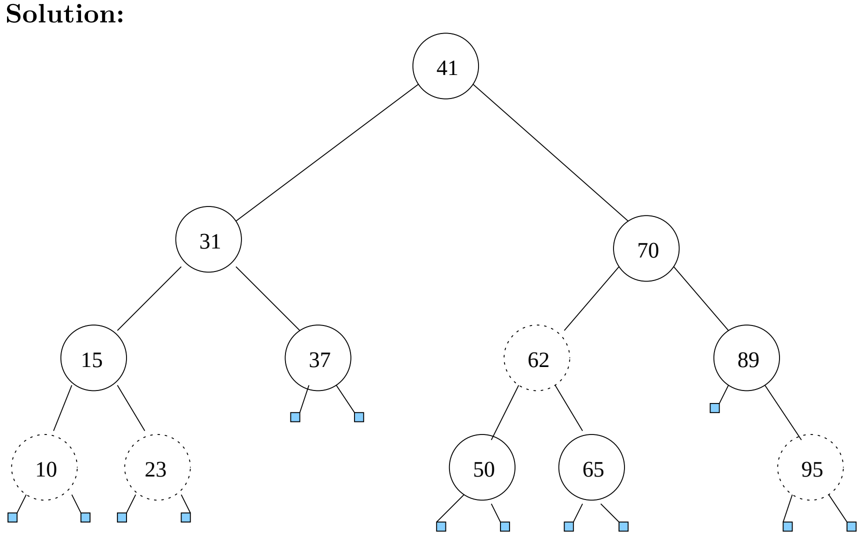
$$1 \cdot \frac{1}{r^2} + 2 \cdot \frac{3(r-1)}{r^2} + 3 \cdot \frac{(r-1)(r-2)}{r^2} = \frac{1 + 6(r-1) + 3(r-1)(r-2)}{r^2} = \frac{3r^2 - 3r + 1}{r^2}.$$

3. Consider the following binary search tree T.

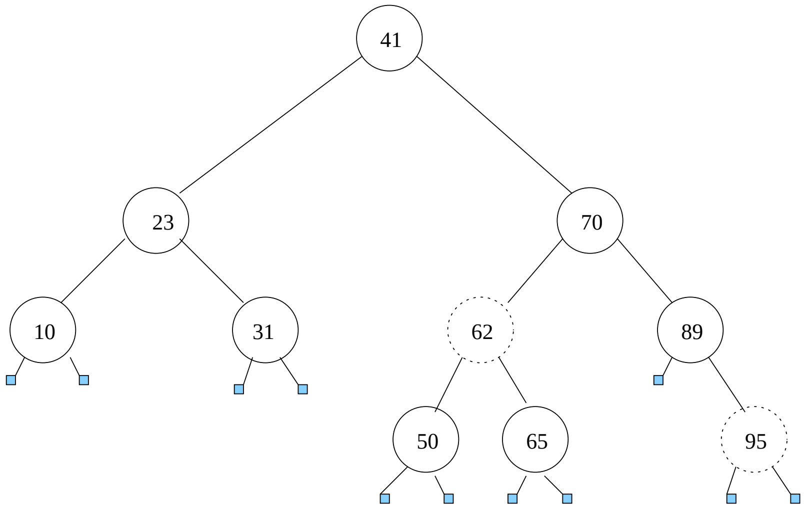


Solid nodes are black, dotted nodes are red.

(a) Draw the red-black tree that results from inserting the key 15 into T.
Solution:



(b) Draw the red-black tree that results from deleting the key 37 from the original tree T.
Solution:



4. Consider a binary tree T. Let $|T|$ be the number of nodes in T. Let x be a node in T, let L_x be the left subtree of x and let R_x be the right subtree of x. We say that x has the "approximately balanced property", $ABP(x)$, if $|R_x| \leq 2|L_x|$ and $|L_x| \leq 2|R_x|$.

   (a) What is the maximum height of a binary tree T on n nodes where $ABP(root)$ holds? Justify your answer.

   Solution:

   The worst case is when L_{root} and R_{root} are just single paths, so that $height(L_{root} = |L_{root}|-1$ (and the same for R_{root}). We know $|L_{root}| + |R_{root}| = n - 1$, so it could be that $|L_{root}| = \frac{1}{3}(n-1)$ and $|R_{root}| = \frac{2}{3}(n-1)$ (or vice versa). Therefore, $height(R_{root}) = \frac{2}{3}(n-1)-1$ and $height(T) = \frac{2}{3}(n-1)$.

   (b) We call T an ABP-tree if $ABP(x)$ holds for every node x in T. Prove that if T is an ABP-tree, then the height of T is $O(\log n)$. More precisely, show that

   $$height(T) \leq \log_2 n / \log_2 \frac{3}{2}$$

   .

   Solution:

   We'll prove that $|T| \geq \frac{3}{2}^{height(T)}$ (*) by induction on the height of T. If T has height 0 (it is a single node), then (*) certainly holds. Now consider T of height h. Assume, without loss of generality, that $height(L_{root}) \geq height(R_{root})$. Then $height(T) = height(L_{root}) + 1$. We know $|T| = |L_{root}| + |R_{root}| + 1$. By $ABP(x)$, this means that $|T| \geq \frac{3}{2}|L_{root}| + 1$. $L_{root} \geq (|frac32)^{h-1}$, so we get $|T| \geq \frac{3}{2}(\frac{3}{2})^{h-1} + 1 \geq (\frac{3}{2})^h$. Now that we hae proven (*), we just take the log of both sides:

   $$height(T) \leq \log_2 n / \log_2 \frac{3}{2}$$

   .

5. Suppose we are given a bit-vector $A = A[1] \ldots A[n]$ of length n (where $A[i]$ is either 0 or 1). We wish to determine if at least half the elements in A are 1's. Consider the following algorithm:

```
HalfOnes( A )
    numOnes ← 0
```

```
numZeros ← 0
for i = 1 to n do
   if A[i] = 1 then
      numOnes++
      if numOnes ≥ n/2 then return true
   else
      numZeros++
      if numZeros > n/2 then return false
```

Measure the complexity by counting the number of array comparisons performed.

(a) What is the best case complexity of HalfOnes? Do not use asymptotic notation. Justify your answer.

Solution: The algorithm can only end if *numOnes* reaches $n/2$ or *numNaughts* exceeds $n/2$, and only one of them is incremented with each iteration of the for loop (and hence with each array comparison).

Since *numOnes* need only reach $n/2$, the best case occurs when the first $\lceil \frac{n}{2} \rceil$ bits are all 1's, giving a running time of $\lceil \frac{n}{2} \rceil$.

(b) What is the worst case complexity of HalfOnes? Do not use asymptotic notation. Justify your answer.

Solution: In the worst case, we need to perform an array comparison for each possible i, giving a running time of n.

This occurs if $A[1] = 0$ and $A[i] = 1 - A[i-1]$ for $2 \leq i \leq n$.

(c) What is the average case complexity of HalfOnes, assuming a uniform distribution? Do not use asymptotic notation. Justify your answer. You may express your answer as a sum.

Remember to formally define the sample space, the probability distribution function, and any necessary random variables, as described in class. You do not need to mathematically simplify your answer.

Solution: Define the sample space for all inputs of size n as $S_n = \{A : A \text{ is a 0-1 vector of length } n\}$
If we assume that the probability of each bit being 1 is $\frac{1}{2}$, each of the 2^n possible bit-vectors in S_n are equally likely.

Let $t_n(A)$ be a random variable represent the number of array comparisons performed on input A. Then $t_n = \begin{cases} \text{position of } \lceil \frac{n}{2} \rceil \text{th } 1 & \text{if } A \text{ has at least half 1's} \\ \text{position of } (\lfloor \frac{n}{2} \rfloor + 1)\text{th } 0 & \text{otherwise} \end{cases}$

The average running time for HalfOnes is

$$\begin{aligned} E[t_n] &= \sum_{A \in S_n} t_n(A) \cdot Pr[A] \\ &= \sum_{i=\lceil \frac{n}{2} \rceil}^{n} i \cdot \frac{1}{2^i} \binom{i-1}{\lceil \frac{n}{2} \rceil - 1} + \sum_{i=\lfloor \frac{n}{2} \rfloor + 1}^{n} i \cdot \frac{1}{2^i} \binom{i-1}{\lfloor \frac{n}{2} \rfloor} \end{aligned}$$

The first term is the summation for the cases where A contains at least half 1's. If the $\lceil \frac{n}{2} \rceil$th 1 occurs in position i, i array comparisons are made; the probability of this happening is the number of ways we can arrange the first $\lceil \frac{n}{2} \rceil - 1$ 1's in the first $i-1$ positions, $\binom{i-1}{\lceil \frac{n}{2} \rceil - 1}$, over all possible bit combinations in the first i positions, 2^i.

Similarly, the second term covers the cases where A does not contain half 1's. If the $\lfloor \frac{n}{2} \rfloor + 1$th 0 (there must be this many 0's) occurs in position i, i comparisons are made, and the probability of this happening is the number of ways to arrange the first $\lfloor \frac{n}{2} \rfloor$ 0's in the first $i-1$ positions, $\binom{i-1}{\lfloor \frac{n}{2} \rfloor}$, over all possible bit combinations in the first i positions, 2^i.

6. We want to augment Red-Black Trees to support the following query, AVERAGE(x), which returns the average key-value in the subtree rooted at node x (including x itself). The query should work in worst-case time $\Theta(1)$.

   (a) What extra information needs to be stored at each node?

   Solution:

   Each node x should store $size(x)$ - the size of the subtree rooted at x - and $sum(x)$ - the sum of all the key values in the subtree rooted at x. The query AVERAGE(x) can be answered in constant time by computing $sum(x)/size(x)$.

   (b) Describe how to modify INSERT to maintain this information, so that its worst-case running time is still $O(\log n)$. Briefly justify your answer.

   Solution:

   Maintaining $size()$ was covered in lecture. Maintaining $sum()$ is exactly the same: when a node x gets inserted, we simply increase $sum(y)$ for every ancestor y of x by the amount $key(x)$.

   Handling rotations for $sum()$ is exactly the same as $size()$ (just replace each $size()$ by $sum()$).

   Hence, INSERT still runs in worst-case time $\Theta(\log n)$.

   (c) Describe how to modify DELETE to maintain this information, so that its worst-case running time is still $O(\log n)$. Briefly justify your answer.

   Solution:

   Again, maintaining $size()$ was covered in lecture. For $sum()$, assume we want to delete node x. If x itself is the node removed, the decrease $sum(y)$ for every ancestor y of x by the amount $key(x)$. If $z = succ(x)$ was removed instead, consider the path from z to the root of the tree. For every node y in between z and x on this path, decrease $sum(y)$ by the amount $key(z)$. For every node y on this path between z and the root (including x itself), decrease $key(y)$ by the amount $key(x)$.

   Hence DELETE still runs in worst-case time $\Theta(\log n)$.

You may find it helpful to implement Red-Black trees using the code from the text, and then modify your code to produce an augmented tree for this problem.