Introduction to Algorithms October 29, 2005
Massachusetts Institute of Technology 6.046J/18.410J
Professors Erik D. Demaine and Charles E. Leiserson Handout 18

Problem Set 4 Solutions

Problem 4-1. Treaps

If we insert a set of n items into a binary search tree using TREE-INSERT, the resulting tree may be horribly unbalanced. As we saw in class, however, we expect randomly built binary search trees to be balanced. (Precisely, a randomly built binary search tree has expected height $O(\lg n)$.) Therefore, if we want to build an expected balanced tree for a fixed set of items, we could randomly permute the items and then insert them in that order into the tree.

What if we do not have all the items at once? If we receive the items one at a time, can we still randomly build a binary search tree out of them?

We will examine a data structure that answers this question in the affirmative. A ***treap*** is a binary search tree with a modified way of ordering the nodes. Figure 1 shows an example of a treap. As usual, each item x in the tree has a key *key*$[x]$. In addition, we assign *priority*$[x]$, which is a random number chosen independently for each x. We assume that all priorities are distinct and also that all keys are distinct. The nodes of the treap are ordered so that (1) the keys obey the binary-search-tree property and (2) the priorities obey the min-heap order property. In other words,

- if v is a left child of u, then *key*$[v] <$ *key*$[u]$;
- if v is a right child of u, then *key*$[v] >$ *key*$[u]$; and
- if v is a child of u, then *priority*$(v) >$ *priority*(u).

(This combination of properties is why the tree is called a "treap": it has features of both a binary search tree and a heap.)

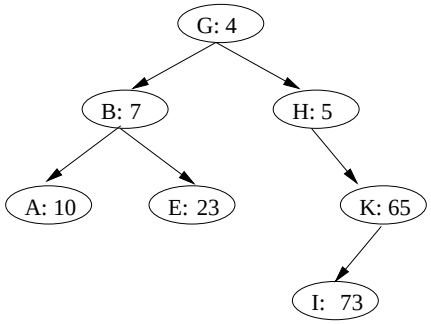

Figure 1: A treap. Each node x is labeled with *key*$[x]$: *priority*$[x]$. For example, the root has key G and priority 4.

It helps to think of treaps in the following way. Suppose that we insert nodes $x_1, x_2, \ldots, x_n$, each with an associated key, into a treap in arbitrary order. Then the resulting treap is the tree that would

have been formed if the nodes had been inserted into a normal binary search tree in the order given by their (randomly chosen) priorities. In other words, *priority*$[x_i] <$ *priority*$[x_j]$ means that x_i is effectively inserted before x_j.

(a) Given a set of nodes $x_1, x_2, \ldots, x_n$ with keys and priorities all distinct, show that there is a unique treap with these nodes.

Solution:

Prove by induction on the number of nodes in the tree. The base case is a tree with zero nodes, which is trivially unique. Assume for induction that treaps with $k-1$ or fewer nodes are unique. We prove that a treap with k nodes is unique. In this treap, the item x with minimum priority must be at the root. The left subtree has items with keys $<$ *key*$[x]$ and the right subtree has items with keys $>$ *key*$[x]$. This uniquely defines the root and both subtrees of the root. Each subtree is a treap of size $\leq k-1$, so they are unique by induction.

Alternatively, one can also prove this by considering a treap in which nodes are inserted in order of their priority. Assume for induction that the treap with the $k-1$ nodes with smallest priority is unique. For $k=0$ this is trivially true. Now consider the treap with the k nodes with smallest priority. Since we know that the structure of a treap is the same as the structure of a binary search tree in which the keys are inserted in increasing priority order, the treap with the k nodes with smallest priority is the same as the treap with the $k-1$ nodes with smallest priority after inserting the k-th node. Since BST insert is a deterministic algorithm, there is only one place the k-th node could be inserted. Therefore the treap with k nodes is also unique, proving the inductive hypothesis.

(b) Show that the expected height of a treap is $O(\lg n)$, and hence the expected time to search for a value in the treap is $O(\lg n)$.

Solution: The idea is to realize that a treap on n nodes is equivalent to a randomly built binary search tree on n nodes. Recall that assigning priorities to nodes as they are inserted into the treap is the same as inserting the n nodes in the increasing order defined by their priorities. So if we assign the priorities randomly, we get a random order of n priorities, which is the same as a random permutation of the n inputs, so we can view this as inserting the n items in random order.

The time to search for an item is $O(h)$ where h is the height of the tree. As we saw in lecture, $E[h] = O(\lg n)$. (The expectation is taken over permutations of the n nodes, i.e., the random choices of the priorities.)

Let us see how to insert a new node x into an existing treap. The first thing we do is assign x a random priority *priority*$[x]$. Then we call the insertion algorithm, which we call TREAP-INSERT, whose operation is illustrated in Figure 2.

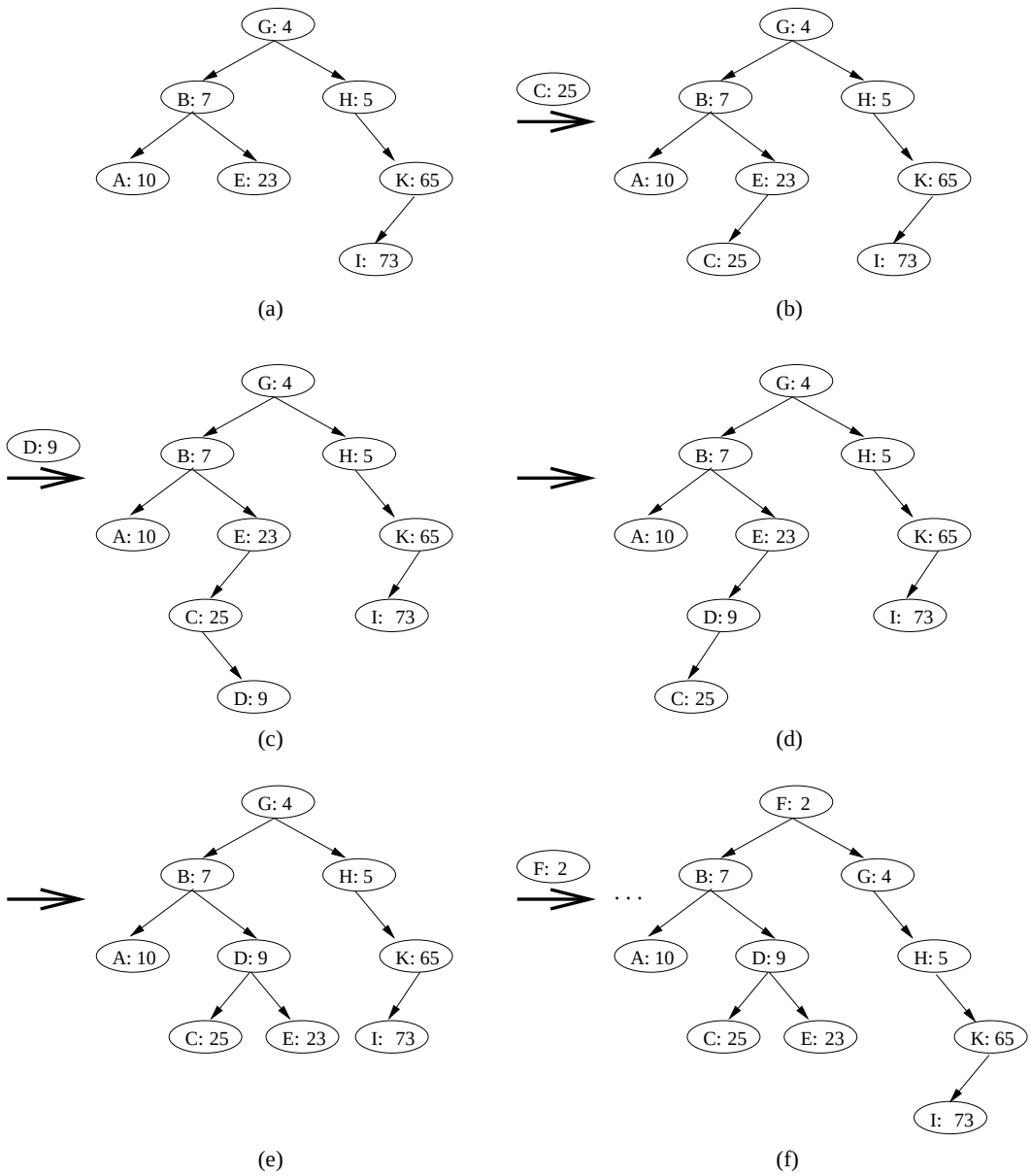

Figure 2: Operation of TREAP-INSERT. As in Figure 1, each node x is labeled with *key*$[x]$: *priority*$[x]$. **(a)** Original treap prior to insertion. **(b)** The treap after inserting a node with key C and priority 25. **(c)–(d)** Intermediate stages when inserting a node with key D and priority 9. **(e)** The treap after insertion of parts (c) and (d) is done. **(f)** The treap after inserting a node with key F and priority 2.

(c) Explain how TREAP-INSERT works. Explain the idea in English and give pseudocode. (*Hint:* Execute the usual binary search tree insert and then perform rotations to restore the min-heap order property.)

Solution: The hint gives the idea: do the usual binary search tree insert and then perform rotations to restore the min-heap order property.

TREAP-INSERT(T, x) inserts x into the treap T (by modifying T). It requires that x has defined key and $priority$ values. We have used the subroutines TREE-INSERT, RIGHT-ROTATE, and RIGHT-ROTATE as defined in CLRS.

```
TREAP-INSERT(T, x)
 1  TREE-INSERT(T, x)
 2  while x ≠ root[T] and priority[x] < priority[p[x]]
 3      do if x = left[p[x]]
 4            then RIGHT-ROTATE(T, p[x])
 5            else LEFT-ROTATE(T, p[x])
```

Note that parent pointers simplify the code but are not necessary. Since we only need to know the parent of each node on the path from the root to x (after the call to TREE-INSERT), we can keep track of these ourselves.

(d) Show that the expected running time of TREAP-INSERT is $O(\lg n)$. **Solution:**

TREAP-INSERT first inserts an item in the tree using the normal binary search tree insert and then performs a number of rotations to restore the min-heap property.

The normal binary-search-tree insertion algorithm TREE-INSERT always places the new item at a new leaf of tree. Therefore the time to insert an item into a treap is proportional to the height of a randomly built binary search tree, which as we saw in lecture is $O(\lg n)$ in expectation.

The maximum number of rotations occurs when the new item receives a priority less than all priorities in the tree. In this case it needs to be rotated from a leaf to the root. An upper bound on the number of rotations is therefore the height of a randomly built binary search tree, which is $O(\lg n)$ in expectation. (We will see that this is a fairly loose bound.) Because each rotation take constant time, the expected time to rotate is $O(\lg n)$.

Thus the expected running time of TREAP-INSERT is $O(\lg n + \lg n) = O(\lg n)$.

TREAP-INSERT performs a search and then a sequence of rotations. Although searching and rotating have the same asymptotic running time, they have different costs in practice. A search reads information from the treap without modifying it, while a rotation changes parent and child pointers within the treap. On most computers, read operations are much faster than write operations. Thus we would like TREAP-INSERT to perform few rotations. We will show that the expected number of rotations performed is bounded by a constant (in fact, less than 2)!

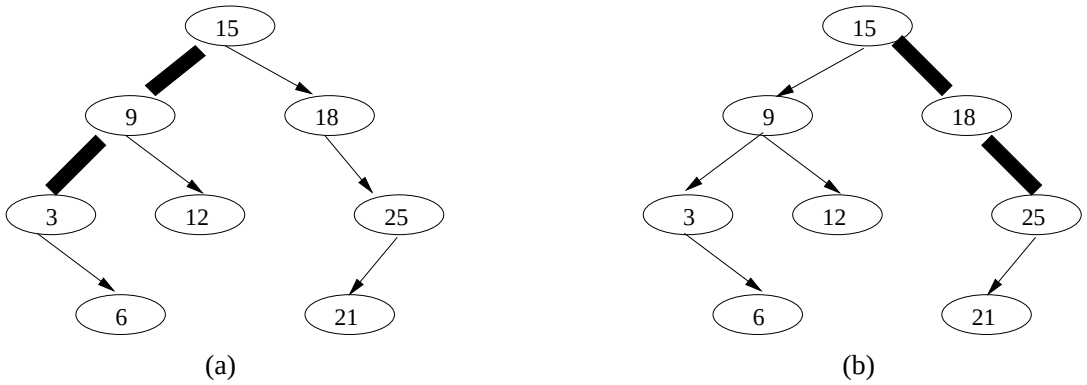

Figure 3: Spines of a binary search tree. The left spine is shaded in **(a)**, and the right spine is shaded in **(b)**.

In order to show this property, we need some definitions, illustrated in Figure 3. The ***left spine*** of a binary search tree T is the path which runs from the root to the item with the smallest key. In other words, the left spine is the maximal path from the root that consists only of left edges. Symmetrically, the ***right spine*** of T is the maximal path from the root consisting only of right edges. The ***length*** of a spine is the number of nodes it contains.

(e) Consider the treap T immediately after x is inserted using TREAP-INSERT. Let C be the length of the right spine of the left subtree of x. Let D be the length of the left spine of the right subtree of x. Prove that the total number of rotations that were performed during the insertion of x is equal to $C + D$.

Solution: Prove the claim by induction on the number of rotations performed. The base case is when x is the parent of y. Performing the rotation so that y is the new root gives y exactly one child, so $C + D = 1$.

Assume for induction that the number of rotations k performed during the insertion of x equals $C + D$. The base case is when 0 rotations are necessary and x is inserted as a leaf. Then $C + D = 0$. To prove the inductive step, we show that if after $k - 1$ rotations $C + D = k - 1$, then after k rotations $C + D = k$. Draw a picture of a left and right rotation and observe that $C + D$ increases by 1 in each case. Let y be the parent of x, and suppose x is a left child of y. After performing a right rotation, y becomes the right child of x, and the previous right child of x becomes the left child of y. That is, the left spine of the right subtree of x before the rotation is tacked on to y, so the length of that spine increases by one. The left subtree of x is unaffected by a right rotation. The case of a left rotation is symmetric. Therefore after one more rotation $C + D$ increases by one and $k = C + D$, proving the inductive hypothesis.

We will now calculate the expected values of C and D. For simplicity, we assume that the keys are $1, 2, \ldots, n$. This assumption is without loss of generality because we only compare keys.

For two distinct nodes x and y, let $k = \textit{key}[x]$ and $i = \textit{key}[y]$, and define the indicator random variable

$$X_{i,k} = \begin{cases} 1 & \text{if } y \text{ is a node on the right spine of the left subtree of } x \text{ (in } T\text{),} \\ 0 & \text{otherwise.} \end{cases}$$

(f) Show that $X_{i,k} = 1$ if and only if (1) $\textit{priority}[y] > \textit{priority}[x]$, (2) $\textit{key}[y] < \textit{key}[x]$, and (3) for every z such that $\textit{key}[y] < \textit{key}[z] < \textit{key}[x]$, we have $\textit{priority}[y] < \textit{priority}[z]$.

Solution:

To prove this statement, we must prove both directions of the "if and only if". First we prove the "if" direction. We prove that if (1) $\textit{priority}[y] > \textit{priority}[x]$, (2) $\textit{key}[y] < \textit{key}[x]$, and (3) for every z such that $\textit{key}[y] < \textit{key}[z] < \textit{key}[x]$ are true, $\textit{priority}[y] < \textit{priority}[z]$, then $X_{i,k} = 1$. We prove this by contradiction. Assume that $X_{i,k} = 0$. That is, assume that y is not on the right spine of the left subtree of x. We show that this leads to a contradiction. If y is not on the right spine of the left subtree of x, it could be in one of three places:

1. Suppose y is in the right subtree of x. This contradicts condition (2) because $\textit{key}[y] < \textit{key}[x]$.
2. Suppose y is not in one of the subtrees of x. Then x and y must share some common ancestor z. Since $\textit{key}[y] < \textit{key}[x]$, we know that y is in the left subtree of z and x is in the right subtree of z and $\textit{key}[y] < \textit{key}[z] < \textit{key}[x]$. Since y is below z in the tree, $\textit{priority}[z] < \textit{priority}[x]$ and $\textit{priority}[z] < \textit{priority}[y]$. This contradicts condition (3).
3. Suppose that y is in the left subtree of x but not on the right spine of the left subtree of x. This implies that there exists some ancestor z of y in the left subtree of x such that y is in the left subtree of z. Hence $\textit{key}[y] < \textit{key}[z] < \textit{key}[x]$. Since z is an ancestor of y, $\textit{priority}[z] < \textit{priority}[y]$, which contradicts condition (3).

All possible cases lead to contradictions, and so $X_{i,k} = 1$.

Now for the "only if" part. We prove that if $X_{i,k} = 1$, then statements (1), (2), and (3) are true. If $X_{i,k} = 1$, then y is in the right spine of the left subtree of x. Since y is in a subtree of x, y must have been inserted after x, so $\textit{priority}[y] > \textit{priority}[x]$, proving (1). Since y is in the left subtree of x, $\textit{key}[y] < \textit{key}[x]$, proving (2). We prove (3) by contradiction: suppose that $X_{i,k} = 1$ and there exists a z such that $\textit{key}[y] < \textit{key}[z] < \textit{key}[x]$ and $\textit{priority}[z] < \textit{priority}[y]$. In other words, z was inserted *before* y. There are three possible cases that satisfy the condition $\textit{key}[z] < \textit{key}[x]$:

1. Suppose z is in the right spine of the left subtree of x. For y to be inserted into the right spine of the left subtree of x, it will have to be inserted into the right subtree of z. Since $\textit{key}[y] < \textit{key}[z]$, this leads to a contradiction.

2. Suppose z is in the left subtree of x but not in the right spine. This implies that z is in the left subtree of some node z' in the right spine of x. Therefore for y to be inserted into the right spine of the left subtree of x, it must be inserted into the right subtree of z'. This leads to a contradiction by reasoning similar to case 1.
3. Suppose that z is not in one of the subtrees of x. Then z and x have a common ancestor z' such that z is in the left subtree of z' and x is in the right subtree of x. This implies *key*$[z] <$ *key*$[z'] <$ *key*$[x]$. Since *key*$[y] <$ *key*$[z] <$ *key*$[z']$, y cannot be inserted into the right subtree of z'. Therefore it cannot be inserted in a subtree of x, which is a contradiction.

Therefore there can be no z such that *key*$[y] <$ *key*$[z] <$ *key*$[x]$ and *priority*$[z] <$ *priority*$[y]$. This proves statement (3). We have proven both the "if" and "only if" directions, proving the claim.

(g) Show that

$$\Pr\{X_{i,k} = 1\} = \frac{(k-i-1)!}{(k-i+1)!} = \frac{1}{(k-i+1)(k-i)}.$$

Solution: We showed in the previous part that $X_{i,k} = 1$ if and only if the priorities of the items between y and x are ordered in a certain way. Since all orderings are equally likely, to calculate the probability we count the number of permutations of priorities that obey this order and divide by the number of total number of priority permutations.

We proved in (e) that whether or not $X_{i,k} = 1$ depends only on the relative ordering of the priorities of y, x, and all z such that *key*$[y] <$ *key*$[z] <$ *key*$[x]$. Since we assumed that the keys of the items come from $\{1, \ldots, n\}$, the keys of the items in question are $i, i+1, i+2, \ldots, k-1, k$. There are $(k-i+1)!$ permutations of the priorities of these items. Of these permutations, the ones for which $X_{i,k} = 1$ are those where i has minimum priority, k has the second smallest priority, and the priorities of the remaining $k-i-1$ items follow in any order. There are $(k-i-1)!$ of these permutations. Thus the probability that we get a "bad" order is $(k-i-1)!/(k-i+1)! = 1/(k-i)(k-i+1)$.

(h) Show that

$$\mathrm{E}[C] = \sum_{j=1}^{k-1} \frac{1}{j(j+1)} = 1 - \frac{1}{k}.$$

Solution:

For a node x with key k, $\mathrm{E}[C]$ is the expected number of nodes in the right spine of the left subtree of x. This equals the sum of the expected value of the random variables $X_{i,k}$ for all i in the tree. Since $X_{i,k} = 0$ for all nodes $i \geq k$, we only need to consider $i < k$.

$$\begin{aligned}
\mathrm{E}[C] &= \sum_{i=1}^{k-1} \mathrm{E}[X_{i,k}] = \mathrm{E}\left[\sum_{i=1}^{k-1} X_{i,k}\right] \\
&= \sum_{i=1}^{k-1} \Pr\{X_{i,k} = 1\} \\
&= \sum_{i=1}^{k-1} \frac{1}{(k-i)(k-i+1)} \\
&= \sum_{j=1}^{k-1} \frac{1}{j(j+1)}
\end{aligned}$$

To simplify this sum, observe that $\frac{1}{j(j+1)} = \frac{j+1-j}{j(j+1)} = \frac{1}{j} - \frac{1}{j+1}$. Therefore the sum telescopes and we have

$$\mathrm{E}[C] = 1 - \frac{1}{k}.$$

If you didn't see this, you could have proven that the equation

$$\sum_{j=1}^{k-1} \frac{1}{j(j+1)} = 1 - \frac{1}{k}$$

holds by induction on k. In the proving the inductive hypothesis you might have discovered $\frac{1}{j} - \frac{1}{j+1} = \frac{1}{j(j+1)}$.

(i) Use a symmetry argument to show that

$$\mathrm{E}[D] = 1 - \frac{1}{n-k+1}.$$

Solution: The idea is to consider the treap produced if the ordering relationship among the keys is reversed. That is, for all items x, leave *priority*$[x]$ unchanged but replace *key*$[x]$ with $n -$ *key*$[x] + 1$.

Let T be the binary tree we get when inserting the items (in increasing order of priority) using the original keys. Once we remap the keys and insert them into a new binary search tree, we get a tree T' whose shape is the mirror image of the shape of T. (reflected left to right). Consider the item x with key k in T and therefore has key $n - k + 1$ in T'. The length of the left spine of x's right subtree in T has become the length of the right spine of x's left subtree in T'. We know by part (g) that the expected length of the right spine of a left subtree of a node y is $1 - 1/idkey[y]$, so the expected length of the right spine of the left subtree of x in T' is $1 - 1/(n-k+1)$, which means that

$$\mathrm{E}[D] = 1 - \frac{1}{n-k+1}.$$

(j) Conclude that the expected number of rotations performed when inserting a node into a treap is less than 2.

Solution:

$$\begin{aligned}\mathrm{E}\left[\text{number of rotations}\right] &= \mathrm{E}\left[C+D\right] = \mathrm{E}\left[C\right] + \mathrm{E}\left[D\right] = 1 - \frac{1}{k} + 1 - \frac{1}{n-k+1} \\ &\leq 1 + 1 = 2.\end{aligned}$$

Problem 4-2. Being balanced

Call a family of trees ***balanced*** if every tree in the family has height $O(\lg n)$, where n is the number of nodes in the tree. (Recall that the ***height*** of a tree is the maximum number of edges along any path from the root of the tree to a leaf of the tree. In particular, the height of a tree with just one node is 0.)

For each property below, determine whether the family of binary trees satisfying that property is balanced. If you answer is "no", provide a counterexample. If your answer is "yes", give a proof (hint: it should be a proof by induction). Remember that being balanced is an *asymptotic* property, so your counterexamples must specify an infinite set of trees in the family, not just one tree.

(a) Every node of the tree is either a leaf or it has two children.

Solution: No. Counterexample is a right chain, with each node having a leaf hanging off to the left

(b) The size of each subtree can be written as $2^k - 1$, where k is an integer (k is *not* the same for each subtree).

Solution: Yes.

Consider any subtree with root r. We know from the condition that the size of this subtree is $2^{k_1} - 1$. We also know that the size of the subtree rooted at the left child of r is $2^{k_2} - 1$, and the size of the subtree rooted at the right child of r is $2^{k_3} - 1$. But the size of the subtree at r is simply the node r together with the nodes in the left and right subtrees. This leads to the equation $2^{k_1} - 1 = 1 + (2^{k_2} - 1) + (2^{k_3} - 1)$, or $2^{k_1} = 2^{k_2} + 2^{k_3}$. Now we show that $k_2 = k_3$. This is easy to see if you consider the binary representations of k_1, k_2, and k_3.

Otherwise, if we assume WLOG that $k_2 \leq k_3$, then we have $2^{k_1 - k_2} = 1 + 2^{k_3 - k_2}$. Now, the only pair of integer powers of 2 that could satisfy this equation are 2^1 and 2^0. Thus $k_3 - k_2 = 0$, or $k_2 = k_3$, and the left and right subtrees of r have an equal number of nodes. It follows that the tree is perfectly balanced.

(c) There is a constant $c > 0$ such that, for each node of the tree, the size of the smaller child subtree of this node is at least c times the size of the larger child subtree.

Solution:
Yes[1]. The proof is by induction. Assume that the two subtrees of x with n nodes in its subtree has two children y and z with subtree sizes n_1 and n_2. By inductive hypothesis, the height of y's subtree is at most $d \lg n_1$ and the height of z's subtree is at most $d \lg n_2$ for some constant d. We now prove that the height of x's subtree is at most $d \lg n$. Assume wlog that $n_1 \geq n_2$. Therefore, by the problem statement, we have $n_2 \geq cn_1$. Therefore, we have $n = n_1 + n_2 + 1 \geq (1 + c)n_1 + 1 \geq (1 + c)n_1$ and the height h of x's subtree is $d \lg n_1 + 1 \leq d \lg(n/(c+1)) + 1 \leq d \lg n - d \lg(1 + c) + 1 \leq d \lg n$ if $d \lg(1 + c) \geq 1$. Therefore, for sufficiently large d, the height of a tree with n nodes is at most $d \lg n$.

(d) There is a constant c such that, for each node of the tree, the heights of its children subtrees differ by at most c.

Solution: Yes[1]. Let $n(h)$ be the minimum number of nodes that a tree of height h that satisfies the stated property can have. We show by induction that $n(h) \geq (1+\alpha)^h - 1$, for some constant $0 < \alpha \leq 1$. We can then conclude that for a tree with n nodes, $h \leq \log_{1+\alpha}(n + 1) = O(\lg n)$.
For the base case, a subtree of height 0 has a single node, and $1 \geq (1 + \alpha)^0 - 1$ for any constant $\alpha \leq 1$.
In the inductive step, assume for all trees of height $k < h$, that the $n(k) \geq (1+\alpha)^k - 1$. Now consider a tree of height h, and look at its two subtrees. We know one subtree must have height $h - 1$, and the other must have height at least $h - 1 - c$. Therefore, we know

$$n(h) \geq n(h - 1) + n(h - 1 - c) + 1.$$

Using the inductive hypothesis, we get

$$\begin{aligned} n(h) &\geq (1 + \alpha)^{h-1} - 1 + (1 + \alpha)^{h-1-c} - 1 + 1 \\ &\geq 2(1 + \alpha)^{h-1-c} - 1. \end{aligned}$$

Suppose we picked α small enough so that $(1 + \alpha) < 2^{1/(c+1)}$. Then $(1 + \alpha)^{c+1} < 2$. Therefore, we get

$$n(h) \geq 2(1 + \alpha)^{h-1-c} - 1 \geq (1 + \alpha)^h - 1.$$

[1]Note that in this problem we assume that a nil pointer is a subtree of size 0, and so a node with only one child has two subtrees, one of which has size 0. If you assume that a node with only one child has only one subtree, then the answer to this problem part is "no".

Therefore, we satisfy the inductive hypothesis.

Note that if we plug this value for α back into $h \leq \log_{1+\alpha}(n+1)$, we get

$$h \leq \frac{\lg(n+1)}{\lg(1+2^{c+1})} \leq (c+1)\lg(n+1).$$

(e) The average depth of a node is $O(\lg n)$. (Recall that the ***depth*** of a node x is the number of edges along the path from the root of the tree to x.)

Solution: No.

Consider a tree with $n - \sqrt{n}$ nodes organized as a complete binary tree, and the other $\sqrt{n}$ nodes sticking out as a chain of length $\sqrt{n}$ from the balanced tree. The height of the tree is $\lg(n - \sqrt{n}) + \sqrt{n} = \Omega(\sqrt{n})$, while the average depth of a node is at most

$$\begin{aligned} & (1/n)\left(\sqrt{n}\cdot(\sqrt{n}+\lg n) + (n-\sqrt{n})\cdot\lg n\right) \\ = \; & (1/n)(n + \sqrt{n}\lg n + n\lg n - \sqrt{n}lgn) \\ = \; & (1/n)(n + n\lg n) \\ = \; & O(\lg n) \end{aligned}$$

Thus, we have a tree with average node depth $O(\lg n)$, but height $\Omega(\sqrt{n})$.