CSC265 Fall 2020 Homework Assignment 7

Solutions

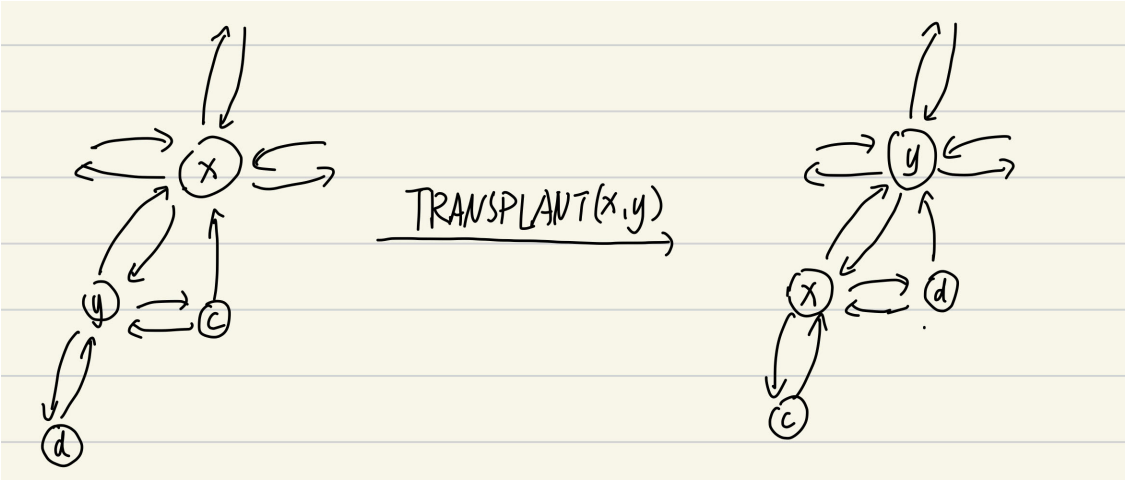
The list of people with whom I discussed this homework assignment: Kunal Chawla

For any node a in our black&white tree, define:

- $a.par$ to be a's parent, or NIL if it does not exist;
- $a.prev$ to be a's sibling on the left (for roots this points to the tree with larger degree), or NIL if it does not exist;
- $a.next$ to be a's sibling on the right (for roots this points to the tree with smaller degree), or NIL if it does not exist;
- $a.first$ to be a's first child (i.e that with the largest degree), or NIL if it does not exist;
- $a.colour$ to be a's colour;
- $a.key$ to be a's key (priority, lower means higher priority);
- $a.degree$ to be a's degree;

We will introduce two constant time operations that we will be using throughout the solution.

The procedure TRANSPLANT(x, y) roughly "swaps" the positions of x and y, where $y = x.first$ and does the following.



In words, $prev, next, par$ of y are updated to x's, with the corresponding "inverse" pointers updated to point to y (for example the inverse pointer of $x.prev$ is $x.prev.next$). In the case where x was not the first child of its parent, there is no inverse pointer of $x.par$, so $x.par.first$ remains unchanged. And $x.par = y$, $y.first$ is updated to x, making d the second child of y (this is by updating $x.next$ and $d.prev$). Now, it is important to updated $x.deg = k, y.deg = k + 1$. Finally, c becomes x's first child (this is by updating $x.first$).

Another procedure we will need is ORGANIZE(x, y), which takes two nodes of equal degree and reorganizes them (when applied to a and a', the number of black nodes strictly decreases). Notice that if x is of degree $l + 1$, we can split x into two nodes of degree l (call them x_1, x_2) in constant time. This is done by:

- "Detach" the first child of x, call it x', by setting $x' = x.first$, and $x.first = x.first.next$;
- Set $x_1 = x$, $x_2 = x'$.

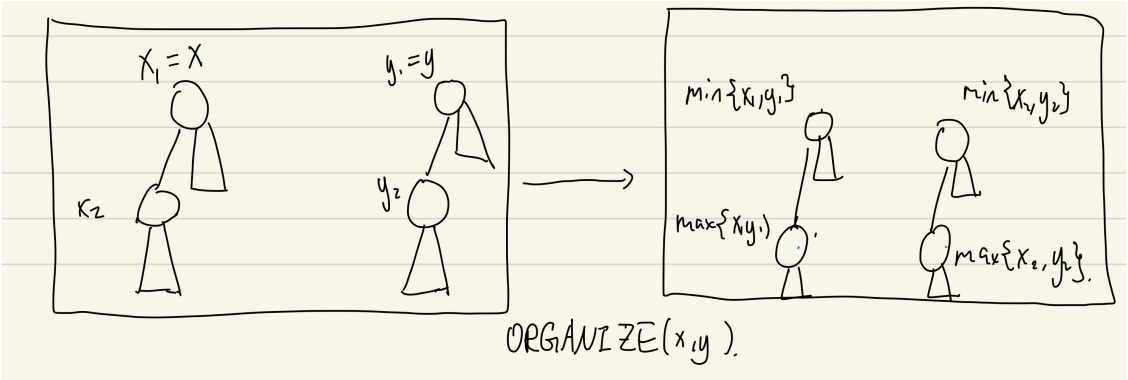
By doing the above, we split x and y into x_1, x_2, y_1, y_2. For x_1, y_1, we call the node with larger key M_1, and the smaller one m_1, and similarly for x_2, y_2 (giving M_2, m_2). The next step of ORGANIZE(x, y) attaches M_1 to m_1, and M_2 to m_2 as the first children. Of course, we will also have to updated the sibling pointers of M_1 and M_2 so that the results are proper binomial trees of degree $l + 1$. Notice now that we can color M_1, M_2 white, since they have keys larger than their parents.

Finally ORGANIZE(x, y) reconnects m_1, m_2 as replacements to x, y. We will keep m_1 connected to its current parent (which is inherited from x or y). This means m_1's colour can be maintained (since from the perspective of m_1's parent, nothing happened to m_1). The last step is to connect m_2 to the remaining node. If m_2 has no parents, set its root to white. If m_2's key is at least that of its parent, set m_2 to white. Otherwise keep m_2 as black.

1. **Solution:** Let the parent of b be a and b' be a'. If $a.key \leq b.key$ and $a'.key \leq b'.key$, then we will simply colour both b, b' white. This will transform H into a black&white heap because there are no black nodes of degree k anymore. And in the case where only one inequality above holds (say for b), we will colour b white. Then, the only property violation is resolved because there are at most one black node of degree k. The other properties are maintained:

   - Roots are still in increasing degrees since we did not change the degree of any node;
   - Every node is still coloured;
   - Roots are still white since b, b' have parents (i.e they are not roots);
   - The new white nodes have keys greater than their parents by above logic and no existing white node has changed;
   - No new black node is produced.

   When $a.key > b.key$ and $a'.key > b'.key$, we will perform ORGANIZE(a, a'):



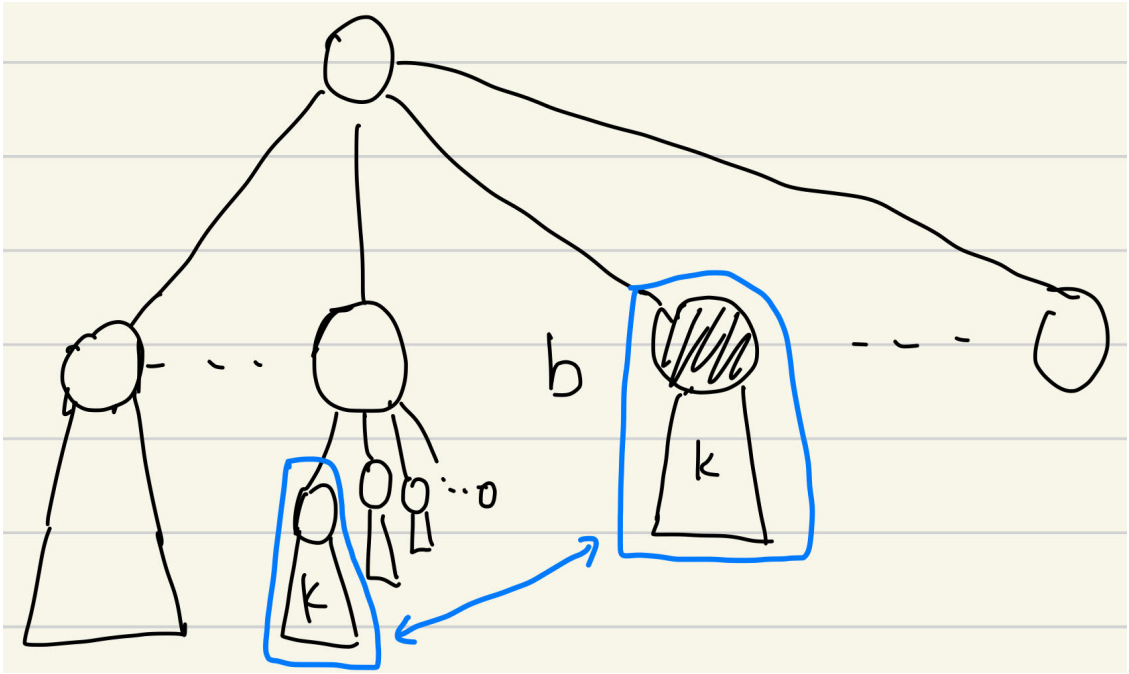
   Note that after the operation, one of b and b' will become white (whichever one that becomes the first child of the other), the other might remain black depending on its relation with its new parent. And for a, a', one of them will retain the colour they had (because the way we reconnect them), and the other (the one in a, a' being the first child of the other) will be white. This means the total number of black nodes in $\{a, b, a', b'\}$ decreases.

The only possible violation of this fixup is that m_2 might be a black node of degree $k+1$ that is not a first child. Also, it is possible that there is already a black node of degree $k+1$. However, H has all other properties of black&white heaps:

- Roots are still in increasing degrees since we did not change the degree of any node;
- Every node is still coloured;
- Roots are still white since if we our operation involved the roots, we made sure that the root is coloured white;
- The new white nodes have keys greater than their parents by above and white nodes that persisted were not operated on;
- All black nodes of degree $< k+1$ remain to have distinct degrees because we did not operate on them nor create any new black node of degree $< k+1$.
- All other black nodes remain first children because we did not operate on them.

2. **Solution:** If $b.par.key \leq b.key$, we will simply colour b white. This fixes the violation that b is black but not a first child, and b being white does not violate anything since $b.par.key \leq b.key$. All other properties that used to hold remains satisfied: no changes have been done to the roots, every node is still coloured, all non-root white nodes have keys at least as large as their parents, and all black nodes are now first children of their parents.

   If $b.par.key > b.key$, we will need a procedure TRANSPLANT'(a, b) similar to TRANSPLANT above. The only difference is that a and b are now disjoint subtrees (i.e that a is not in b and b is not in a). To carry out this procedure, we only need to swap the pointers $prev, next, par$ of a and b, and the corresponding inverse pointers. In particular, we will perform TRANSPLANT'$(a.first, b)$, where $a = b.prev$:



   Here, we make some helpful observations. First, notice that $a.first$ must be white since it is of degree k and b is the only black tree of degree k. Second, because both a and $a.first$ are

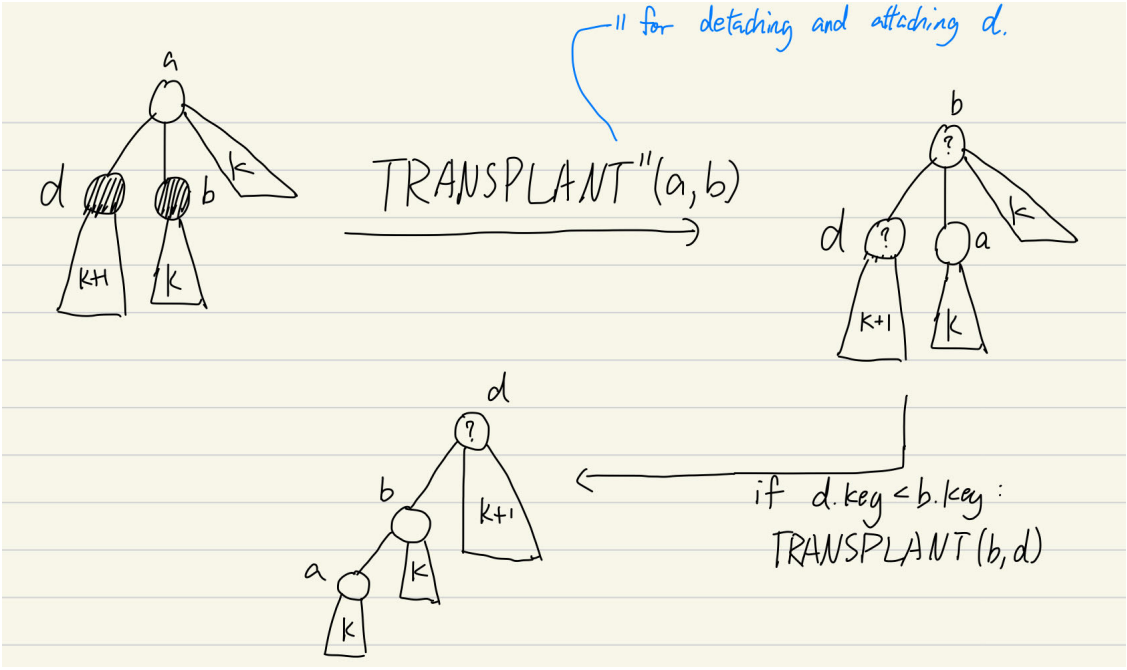
white, we have the inequality $a.first.key \geq a.key \geq a.par.key$. This means $a.first$ is justified to be in the position of b as a white node. The resulting data structure is a black&white binary heap because:

- Roots are still in increasing degrees since we did not change the degree of any node;
- Every node is still coloured;
- Roots are still white since we did not change roots;
- $a.first$ being a white child of b is justified since $a.first.key \geq b.par.key$;
- No new black nodes have been produced;
- b being black is justified since it is now the first child of its parent and it has degree k, which is one less than its parent of degree $k+1$;

3. **Solution:** Let $d = b.prev$, $a = b.par$. If we can simply colour d white (i.e that $a.key \leq d.key$), this becomes what we handled in the previous question and we will simply call on the procedure defined previously. Thus, we will assume that $a.key > d.key$ for this question.

   If b has key greater than or equal to that of its parent, then we will simply colour b white. This will make sure H is a black&white heap since the only violation is resolved (we no longer have a black node that is not a first child), and the new white node has key greater than or equal to that of its parent. All other properties remain to hold since no changes relevant to them has been applied.

   Now, suppose that $b.key < b.par.key$. We will find a way to resolve this that decreases the number of black nodes as in question 1. Notice that since d is black and black nodes are first children, we know d is the greatest sibling of b and a is of degree $k+2$. Notice that if we disconnect d temporarily from a (as in the way we split x in the definition of ORGANIZE(x, y)), then a is a binomial tree of degree $k+1$. With this, we will apply TRANSPLANT(a, b). Since $b.key < b.par.key$, we know can colour a white. Reattaching (again, "reattachment" is defined in ORGANIZE(x, y)) d to b, we obtain a degree $k+2$ tree b with $b.first = d$. If $d.key \geq b.key$, we will set d to white, thereby achieving our goal of decrease the black node count. However, if that is not the case, we will perform TRANSPLANT(b, d) and colour b white. Let r be the root of our current tree of degree $k+2$ (we know $r \in \{a, b, d\}$). Finally, we will colour r depending on how its key compares to that of its parents (white if greater than or equal to or this is the root, black otherwise).



The only violation of our solution happens when r is black and there is already a black node of degree $k+2$, or when r is black and it is not the first child of its parent. All other properties of a black&white node are satisfied:

- Roots are still in increasing degrees since we did not change the degree of any node;
- Every node is still coloured;
- Roots are still white since if we updated the root, we made sure it is white;
- a, b, d are properly coloured. These are the only new white nodes introduced and no other existing white node changed their parents. Thus all white non-root nodes have keys at least that of their parents;
- For $k' < k+2$, there is at most one black node of degree k'' and it must be a first child because we did not create any black node of degree less than $k+2$ (the previous culprit b has been changed).

4. **Solution:** Let a, a' be the parents of b, b', respectively.

   If $b'.key \geq a'.key$, we colour b' white and call procedures define in 2 or 3 depending on the colour of $b.prev$. This is justified since if we can make b' white, then b is the unique black node with degree k, so we can apply procedure in 2 or 3. After this we can return.

   If $b'.key \geq a.key$, we simply perform TRANSPLANT'(b', b), colour b' white (this is legal since $b'.key \geq a.key$), and return.

   If $b.key \geq a.key$, we colour b white and return.

   If $b.key \geq a'.key$, we perform TRANSPLANT'(b', b), colour b white (we can do this since $b.key \geq a'.key$), and apply procedure 2 or 3 for the same reason above in the first case.

   Notice that in all cases above, the number of black nodes decrease.

~~In the case where none of the inequalities hold above, we know that $\max\{b.key, b'.key\} < \min\{a.key, a'.key\}$.~~
RIP MY MARK

5. **Solution:** We will augment the black&white heap H to contain a doubly-linked list lk of pointers to black nodes, sorted from lowest to highest degrees. Initially this is empty. The augmented data structure will also contain a dynamic array arr of size at least $2\lceil \log_2 n \rceil$, where n is the number of nodes in H. After each heap operation, $arr[i]$ stores a pointer to a node in lk, which is a pointer to a degree i black node (if such a node does not exists, $arr[i]$ is NIL). Notice that accessing a linked list is constant time. Array arr is initialized to NIL and its size grows or shrinks depending on n. Adding or deleting a black node is constant time since arr has size $2\lceil \log_2 n \rceil$, which is the maximum degree of H. Also, maintaining that arr has size $2\lceil \log_2 n \rceil$ has amortized time complexity $O(1)$, so this will not affect the efficiency of H.

   We now explain how to perform DECREASE-KEY(H, x, v) in amortized constant time. Given the ground work of questions 1-4, this is not difficult:

   (a) Set $x.key = x.key - v$;

   (b) Check if there is a black node of degree $k = x.deg$. Check if $x.par.key > x.key + 1$.
   If there is no other black node of the same degree and $x.par.key = x.key + 1$, we return since there is no violation.
   If there is no other black node of the same degree, but $x.par.key > x.key + 1$: check $x.prev.colour$ and execute procedure defined in question 2 or 3 depending on the colour. If procedure in question 2 is executed, the resulting heap is valid and we return. If procedure in question 3 is executed, the resulting heap's validity depend on the colour of r (recall that r is the root of the tree with degree $k+2$). Since the first and second children of r are white, we can remove $arr[k+1]$ from lk and set $arr[k+1]$ to NIL. Now, if r is black, we will add r to our linked list and dynamic array. Finally, we execute (b) with $x = r$.
   If there is another black node of the same degree, and $x.par.key = x.key + 1$: we call procedure defined in question 1, during which if a node is changed from black to white and stays so, we remove it from our array and linked list. Finally, we execute (b) with $x = m_2$.
   If there is another black node of the same degree, and $x.par.key > x.key + 1$: we will call procedure defined in 4, during which if a node is changed from black to white and stays so, we remove it from our array and linked list. Use the above methods for calling (b) for possible violations if 4 in turns calls 2 and 3. If no calls to 2 or 3 is made by 4, we execute (b) with $x = r$.

   Let us analyze the amortized time complexity of DECREASE-KEY. Let Φ be our potential function, where $\Phi(H)$ is the number of black nodes in H. We can see that the actual cost of DECREASE-KEY is (1 + the number of times (b) is called), and every time (b) is called, one of procedure 1-4 is called (calling 2,3 in 4 counts as one), costing us 1. But notice that a call to 1,2, or 4 decreases $\Phi(H)$ by at least 1.

   Therefore, the amortized complexity DECREASE-KEY is

$$\hat{a} = (\text{the number of times (3) is called}) + \Phi(H_f) - \Phi(H_i) \leq 2,$$

since call to 2 does not decrease $\Phi(H)$ but occurs at most once.