EECS 4101

ASSIGNMENT 3

MICHAEL WILLIAMS

211087798

APRIL 3, 2017

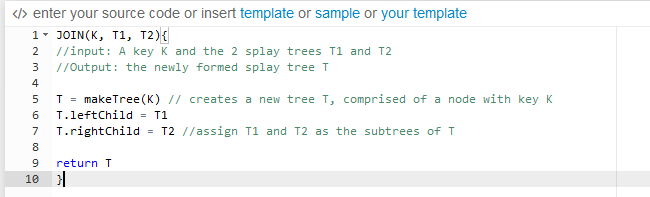
**The following assignment was collaborated with**

**Richard Bun, student number: 211119732**

***Question 1***

**Part A:** For this question, we are required to design algorithms for the functions split and join, that are as efficient, in an amortized sense, as possible. In addition, these functions should not affect the amortized time complexities of the other splay tree operations.

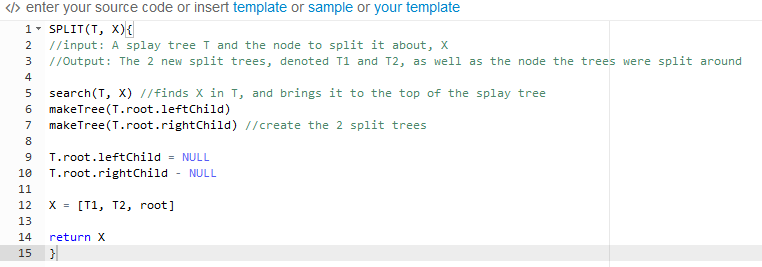
Below is the algorithm for join. Given a key K, and two splay trees T1 and T2, the algorithm simply creates a new splay tree whose root is a node with key K, and whose left and right sub trees are T1 and T2 respectively. All keys of T1 are less than K and all keys of T2 are greater than K.



The algorithm works as such: first, a new splay tree is created and rooted at the node K. This splay tree is denoted T. Then, we assign T’s left child to be the root of T1 and T’s right child to be the root of T2. Then we return T. We know this design must be correct, because the resulting tree is rooted at a node with key K, whose left and right sub trees are T1 and T2 respectively. More so, the resulting splay tree does not violate any splay tree properties.

Because implementing the function does not involve creating new attributes, or a need to update the attributes of the resulting splay tree, the amortized complexities of the other splay operations does not change in terms of functionality or amortized efficiency. The amortized time for join is O(log n) in the worst case. This is explained in part B.

Next, we look at the algorithm for split. Given a splay tree T and a node X in T, the algorithm creates splay trees T1 and T2. T1 consists of all elements in T whose keys are less than X.key and T2 consists of all elements in T whose keys are greater than X.key. The function itself returns both the sub trees T1 and T2, in addition to the rooted node, ie the node around which T1 and T2 were split.



The algorithm works as such: first, we call the search function to find the node X. We assume that the search function performs the necessary splay steps to maintain the properties of a splay tree and that both the search and the splay perform as described in the lecture slides and LN4. This means that after the function has been called, T’s root should be node X and the tree itself shouldn’t violate any properties. Next, we create two new trees, T1 and T2, who are rooted at the left and right sub trees of T, respectively. The nodes of T1 are guaranteed to be less than X.key and the nodes of T2 are guaranteed to be greater than X.key. Last, we cut all connections from the root of T, to the sub trees T1 and T2. Finally, we return T1, T2, and the now single node splay tree, T, as specified.

Because implementing the function does not involve creating new attributes, or the need to update the attributes of the resulting splay tree, the amortized complexities of the other splay operations does not change in terms of functionality or amortized efficiency. The amortized time for split is O(log n) in the worst case. This is explained in part B.

**Part B:** For this question, we must show that the approximated amortized time bounds described above are correct. Before analyzing the functions, we must establish a few things.

First, the amortized time of each operation is described by the equation

(C)

Where C is the actual cost and Φ (T) and Φ (T’) represent the potential of the splay trees before and after the call to a function, respectively.

Next, we recall form *LN3* that the potential of a splay tree is a function of its rank. More specifically,

(D)

We also know that the rank of a node is a function of its weight, or more specifically

(E)

Where the weight is defined as the number of nodes in the subtree rooted at u. Thus

(F)

We can now start by analyzing the join function. The actual cost is clearly constant O(1) time, as only a constant amount of work is being performed, every time it is called. The potential prior to the function call is

(G)

The potential after the function call still consists of the same trees T1 and T2, only they are rooted at the new node K, so the potential of T1 and T2 stay the same. However their parent node K now has a potential of at most log(n), assuming n is the number of nodes in the splay tree, as explained below:

(I)

Thus the potential after the function call is

(J)

Overall, the amortized cost is

(K)

And therefore, the amortized cost is O(log n) in the worst case.

Next, we analyze the split function. The search takes O(log n) time and the splay takes an additional O(3 log n + 1) time. Aside from that, only a constant amount of work is being done each call, so the remainder of the function takes constant O(1) time. Overall, the actual cost of the split function is O(4log n + 1) time in the worst case. The potential prior to the call to split is simply the potential of the splay tree T, or Φ(T).

The call to split does not increase potential at all. Consider the splay tree T’ after the search function has been called. Although it has been rearranged, the tree is still comprised of the same nodes as before. More so, we know from the lecture slides and *LN3* that the new tree still retains the same potential, because its weight and rank are unchanged. After the call to search, the remainder of function simply breaks up the tree into 3 parts: Its root, left child and right child. The left and right sub trees, T1 and T2 still retain the same potential, but the root’s potential now changes to 0, as seen below

(L)

Thus, the new potential is actually smaller than the potential prior to the call to split, and thus the change in potential is 0.

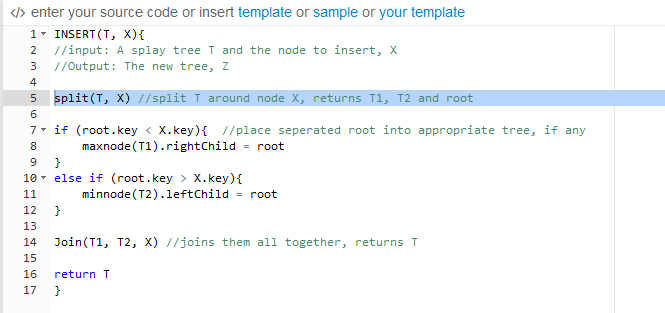
Overall then, the amortized cost of the split function is

(M)

And therefore, the amortized cost is O (log n).

**Part C:** For this question, we describe how to implement the splay operations insert and delete efficiently, using the split and join operations described above. We then give the amortized cost bound for each, assuming that the costs of the functions described above are correct.

We begin with the insert operation. The algorithm below shows how it would be implemented in O (log n) time, in the worst case.



The algorithm works as such: first, we split the tree T at X.key. Although the split function expects that the node exists when used by itself, the insert function still works for this implementation. If the node we wish to insert already exists in T, then the split works as expected. But if the node we wish to insert doesn’t exist in T, the split function will split T around either X’s predecessor or successor node. The if statements that follow effectively check the value of the new root (ie the value T was split at), compare it to the value of X.key, and place it in its appropriate sub tree, if it goes into a sub tree at all. Last, the function joins the two sub trees T1 and T2 to the new node X and returns the newly created tree.

The actual cost of the function is O (log n) time in the worst case. The code simply calls the functions join and split and the if statements take at most O(log n) time to find the max or min node in T1 or T2 respectively. Adding this additional node costs no extra potential, as it is guaranteed to be a leaf node. Therefore, its rank, as shown below, is 0

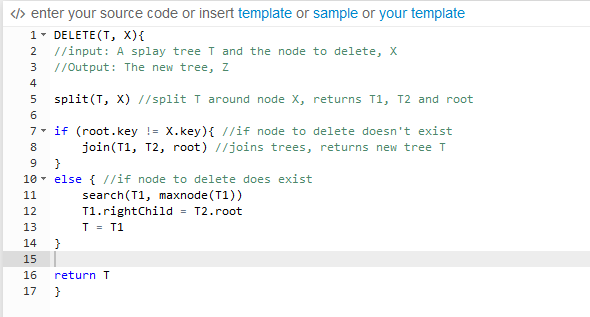
(O)

Furthermore, we already know that the calls to join and split both have an amortized cost of O (logn) in the worst case. Therefore, the total amortized cost of the new search function is:

(P)

And so, the amortized cost of the function is O (log n).

Next, we look at the delete operation. The algorithm below shows how it would be implemented in O (log n) time, in the worst case.



The algorithm works as such: first, we split the tree T around the node you wish to delete, X. Then, we check to see if the node T was split around was actually node X. In other words, we check to see if X actually existed in T. If it didn’t, the first if statement joins the trees back together with the join function and assigns Z to the newly created tree. However, if X does exist in T, we search for the largest key in the Left sub tree, T1. After the search and splay operations, the largest key of T1 will now be at the root. As a result, it will have no right child node, as there can be no larger keys in the sub tree. Then, we can simply connect T2, whose values are all greater than T1’s root, as its right child, and return the newly formed tree.

The actual cost of the function is O (log n) in the worst case. All the normal code execution, aside from the calls to join and split is done in constant O(1) time. However, when X exists in the splay tree, the search for the largest value in the left sub tree T1, takes O(log n) time. The additional splay operations that follow cost at most 3log n + 1 time. Regardless of the outcome, the split function is always run, and has an amortized cost of at most O(log n). Furthermore, there is never an increase in potential. We can analyze this by considering both cases:

*Case A:* If the node we wish to delete doesn’t exist in T, the tree is simply rejoined, with the join function, and returned. Join has an amortized cost of O(log n) in the worst case, and we know there is no change in the potential, from *LN3* and the lecture slides, or more specifically, because the tree remains the same, only rearranged, similar to a search function. Such a function doesn’t change the potential of the tree, so it shouldn’t change in this case either.

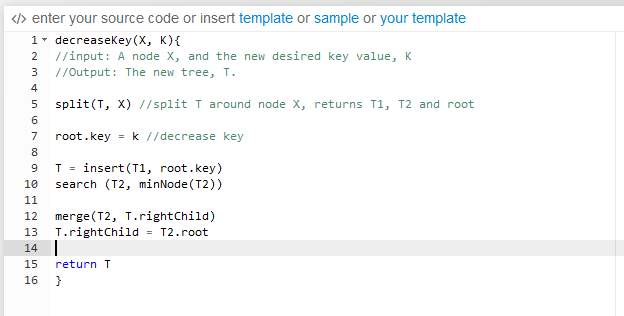
*Case B:* If the node we wish to delete does indeed exist in T, a search is first done for the largest node in the splay tree T1. The potential does not increase, for the same reasons described above, in the analysis of the split function and in Case A: Although the splay tree has been rearranged, T1 is still comprised of the same nodes as before. Furthermore, the root won’t be included in the returned tree, so the total potential will actually decrease.

Thus, with no increase in potential in either case, the amortized cost of the delete function, in the worst case, is

(R)

**Part D:** For this question, we describe how to implement the function decreaseKey(x, k) efficiently for a node X in a splay tree and show its amortized cost bound. decreaseKey(x, k) takes the node X in the tree, and reduces the key of the node *X* to the value *k*, where *k* is guaranteed to be less than or equal to the original value of *x.key*. For this function, we assume that node X is in the tree T when the function is called.

The algorithm would be implemented as shown below, in O(log n) worst case time.



The algorithm works as such: First, we split the tree around X with the split function. This will create the trees T1, T2, and the tree comprised of just node X. Then, we decrease the value of node X. Now, all that’s left is to recombine the trees together in a way that satisfies a splay tree. We begin by inserting the newly decreased node X into T1, using insert (T1, X.key), as described above. This will create a new tree, that we call Z. The tree Z has at its root, the new X node, and in its left and right sub trees, all the nodes of T1 that are less than X.key, and greater than X.key, respectively. We then move away from this tree for a moment, and use the search function of splay trees to look for the minimum node of T2 using findMin(T2). The resulting tree T2 contains at its root, the minimum key, and on its right, all the other nodes of the tree. Because no smaller nodes exist, the left sub tree is empty. Now, we must connect T2 and Z together. To do so, we first merge Z’s right child, with T2, while making sure to cut off any connections to Z. Because all the values of Z are still smaller than any value of T2, this entire sub tree will merge onto the left child of T2. Now, we simply merge the root of T2 with Z’s now empty right sub tree. Now, we have a complete splay tree, in which the decreased key is at the root, that doesn’t violate any splay tree rules.

In terms of the amortized cost, this function takes O (log n) time in the worst case. The actual cost of the function is constant O (1) time, except for the search function, which costs O(log n) time, as described in the lecture slides. Merge is simply a reconnecting of trees to different parents, which takes constant O(1) time. Thus, the actual cost of this operation is O (log n) in the worst case. Next, In terms of potential, nothing has changed. The new tree still consists of the same nodes as the original tree. Only the order has been changed. That being said, this means that the change in potential differs by as much as it would have, had a simple search function been run. In other words, the only cost applicable is the actual cost, and the potential remains the same. However, it is worth mentioning that the split and insert function runs at an amortized cost of O(log n). Thus, the total amortized cost of this function is:

(T)

Thus, the amortized cost is bounded by O(log n) in the worst case.

**Part E:** For this question, we discuss the difficulties with implementing the merge or union operation efficiently. The biggest problem when implementing this comes from the instances when both splay trees already contain two child nodes at its root. With two child nodes already present in each tree, the program must separate them and then rejoin them in such a way that does not violate the properties of a splay tree. More so, it must be done in efficient time. This was already difficult when implementing the decreasekey function, which only involved the use of 1 tree, although it was implemented in such a way where the amortized cost only increased by a constant amount. This problem becomes much more difficult when dealing with 2 trees. The general means to implement such a function efficiently is to isolate the node that will become the new root, and then figure out a way to separate the trees together in a way in which they can be rejoined, while still following the properties of a splay tree, and still holding the properties of the designated root of the new splay tree.

***Question 2***

**Part A:** Argue that line 10 executes exactly once for each pair of nodes {u, v} that exist in P.

Looking at the code, you can see that each pair in P will be considered twice in the program, as a result of the tree traversal being done as a depth first search algorithm and nodes being coloured to black in a post order fashion. More specifically, the pair will be considered each time a member of the pair is coloured black. The first time the pair is considered is after 1 of the nodes is coloured black. Because only one of the two nodes is coloured black, line 10 won’t be executed. The second time is when the other node has been coloured black. At this point, because both nodes are coloured black, line 10 will execute. After this, the code will never consider that specific {u,v} pair again. Thus, line 10 will only execute exactly once for each pair of nodes {u,v} that exist in P.

**Part B:** Argue that at the time of the call LCA(u), the number of sets in the disjoint-set data structure equals the depth of U in T

We will argue that this is true through induction. First, we consider the 3 base cases, in which we have a Tree T, with 0, 1 and 2 nodes. It is very clear that in the case of a set with no nodes or 1 node, nothing will be printed. In a set with 2 nodes, u and v, in which v is a child of u, the common ancestor of the two is node u, and thus, line 10 will be executed.

Now, we consider the inductive hypothesis. Assume that the number of sets in the disjoint-set data structure equals the depth of U in T, prior to a call of LCA. More specifically, assume that prior to the call, node u has a depth of X and that there are X items in the disjoint-set data structure.

Now, we show that after the next call to LCA, the number of sets in the disjoint-set data structure still equals the depth of the node called through LCA.

At the beginning of this call in line 1, the number of disjoint sets increases to X + 1. After the LCA function calls a child node, there will be X + 2 disjoint sets, but the union function reduces it back to X + 1 disjoint sets for the remainder of the for loop. Afterwards, there are no changes in the number of disjoint sets, so the algorithm ends with X + 1 disjoint sets, which is equal to the depth of the new node in T.

Because we’ve proven that the hypothesis holds true at the end of the call to LCA, this means, that the number of sets in the disjoint-set data structure equals the depth of a node U in T prior to a call to LCA(u).

**Part C:** Prove that LCA correctly prints the least common ancestor of u and v for each pair {u,v} that exists in P.

We start by denoting the LCA of two nodes u and v to be X. Because the algorithm traverses in the form of a depth first search, we know that the algorithm returns to node X after the call to node v. We also know that if LCA(u) is returned, then u is in the same set as its parent, as a result of the union function in line 5. With this, we can confirm that v and X are in the same set. The node would not have returned from node X, because u has not yet been returned. We also know that if the function LCA(u) has been called but not returned, then ancestor[find(u)] = u, because it is set to itself initially, and will set ancestor[find(u)] to u, repeatedly, after a union operation is performed. Applying this, we know that the ancestor[find(X)] will still be X. Thus, the LCA will correctly print out the least common ancestor of u and v, or more specifically, ancestor[find(v)] will be X.

**Part D:** Analyze the running time of LCA.

In analyzing the running time of LCA, we break up the algorithm into 2 parts. The first part will be up to line 7, where the node is coloured black, and the second part will be the remainder of the algorithm, where the statement is possibly printed out. Assuming that the tree T contains n nodes, LCA will be run n times.

In these n runs, most of the first part of the algorithm will be executed in constant O(1) time. In addition to this, m disjoint set operations are conducted on the n elements of the tree, where m represents the number of pairs in P. Thus, over the run of n nodes, this means that the running time will be O(n + m). However, this runtime does not consider the union and find operation times. From the textbook, we know that the combined union by rank and path compression heuristics cost and additional **α** time, for each of the m disjoint set operations on the n elements of the tree. Thus, the total time for the first part of the algorithm is O[(n + m) **α** (n)].

In the second part of the algorithm, each pair is checked twice, but the print statement in line 10 is only executed once for each pair, as explained in part A. m pairs are checked in P for every node in the tree T. Thus, the second part of the algorithm takes O(nm) time.

Therefore, the total complexity of the LCA algorithm on a splay tree T of n nodes is

(U)

: