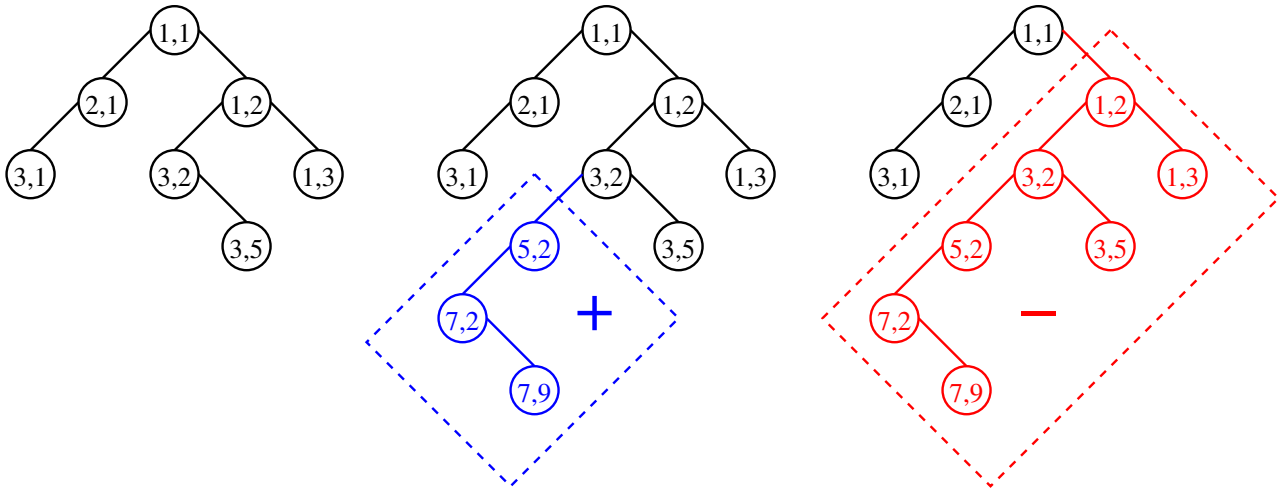
CS69001 Computing Laboratory – I

Assignment No: B2

Date: 14–August–2019

This assignment deals with a particular type of binary tree T. Each node in T stores a pair (a,b) of positive integers. At the root, this pair is $(1,1)$. Let the node v store the pair (a,b). Then, the left child of v, if it exists, stores the pair $(a+b,b)$, and the right child of v, if it exists, stores the pair $(a,a+b)$. Since $\gcd(1,1)=1$, it follows by induction that the pair (a,b) stored at any node satisfies $\gcd(a,b)=1$. Conversely, any pair (a,b) of positive integers with $\gcd(a,b)=1$ is a valid pair for storage at a unique node of T. Part (a) of the following figure shows an example of a pair tree T.



(a) A pair tree (b) Effect of insert(7,9) (c) Effect of delete(1,2)

Declare a suitable data type to store a node of T. Each node should consist of a, b, a left-child pointer L, a right-child pointer R, and nothing else. Initialize T to a tree of a single node (the root) storing the pair $(1,1)$. We assume that this node always remains in T.

Part 1: Searching in T

Suppose that we want to search for a pair (a,b) in T. If $\gcd(a,b)\neq 1$, this pair cannot be present in T, and the search fails immediately. So suppose that $\gcd(a,b)=1$.

Unlike binary search trees, the search proceeds up the tree. If $a>b$, the node (if any) storing this pair must be the left child of a node storing $(a-b,b)$. On the other hand, if $a<b$, the node storing this pair is the right child of node storing $(a,b-a)$. Proceeding this way, you eventually end up at the root $(1,1)$.

Make a linked list L of the pairs on the search path. Insert pairs at the beginning of the list, so the list starts with $(1,1)$, and ends with (a,b). Now, start from the root of T, and traverse T and L simultaneously. The traversal in T (that is, whether you go to the left child or the right child of the current tree node in T) is guided by the linked list L. If you encounter a NULL pointer in T before the traversal completes (that is, L ends), the search fails. If the traversal completes successfully, return a pointer to the node of T, at which the traversal ends. Write a function *tsearch* to implement this algorithm.

Before the function returns, make sure to free the memory allocated to all the nodes in the linked list L.

Part 2: Insertion of a pair in T

Suppose that we want to insert a pair (a,b) in the tree T. If $\gcd(a,b)\neq 1$, this is not a valid pair for T, so make no effort to insert this pair. Otherwise, proceed as follows.

If (a,b) is already present at some node of T, nothing needs to be done. Otherwise, a new node for storing (a,b) is to be created, and appropriately linked to its parent node. The parent of (a,b) is the node storing the unique pair $(a-b,b)$ or $(a,b-a)$ depending upon whether $a>b$ or $a<b$. But this parent node too may

be not present in T, so the parent node needs to be inserted too. The parent of the parent may again be not present in T, so another new node is to be inserted. This process will go on so long as necessary. Eventually, you reach the root $(1,1)$ which always exists in T. Part (b) of the above figure shows the insertion of the pair $(7,9)$ in the tree of Part (a). This insertion calls for the addition of three new nodes.

The implementation of the insertion procedure should not proceed from higher to lower levels of the tree. This way of doing it is too costly. As in the case of search, create a linked list L of pairs, storing the unique path from $(1,1)$ to (a,b). Start a simultaneous traversal of L (from the beginning) and T (from the root). So long as the traversal continues, no new nodes need to be created. In particular, if the traversal completes successfully, (a,b) is already present in T, and nothing needs to be done. If the traversal in T encounters a null pointer at some point, all the remaining nodes on the path need to be created and appropriately linked. Write a function *tinsert* to implement this idea.

Again, do not forget to free the memory associated with L before the function returns.

Part 3: Deletion of a pair from T

Suppose that we want to delete the pair (a,b) from T. First, check whether $\gcd(a,b) = 1$. If not, this pair is not present in T. Otherwise, proceed as follows. If $(a,b) = (1,1)$, do nothing, for the root should never be deleted. So suppose that $(a,b) \neq (1,1)$. Make a search for (a,b) in T. If the search fails, do nothing.

If (a,b) resides in an internal (that is, non-leaf) node, the deletion of this node makes its child/children orphan(s). This is not allowed. So the rule is that the entire subtree rooted at the node storing (a,b) is to be deleted. Moreover, since this is not the root node, the appropriate child pointer of the parent node should be set to null. See Part (c) of the figure on the last page for an example.

Write a function *tdelete* to implement this deletion algorithm. The function must free the memory associated with every deleted node.

Part 4: Printing the tree T

Write three functions for printing the pairs stored in the tree T: *preorder*, *inorder*, and *lexorder*. The first two functions are straightforward. The third function is non-trivial, and is explained now.

Let (a,b) and (c,d) be two pairs. We say that (a,b) is lexically or lexicographically smaller than (c,d) if either $(a < c)$ or $(a = c$ and $b < d)$. The function *lexorder* is meant for printing the keys stored in T in the lexicographically sorted order. Neither of preorder, inorder, postorder, and level-by-level traversals can do this printing. You need to implement a special algorithm for this task.

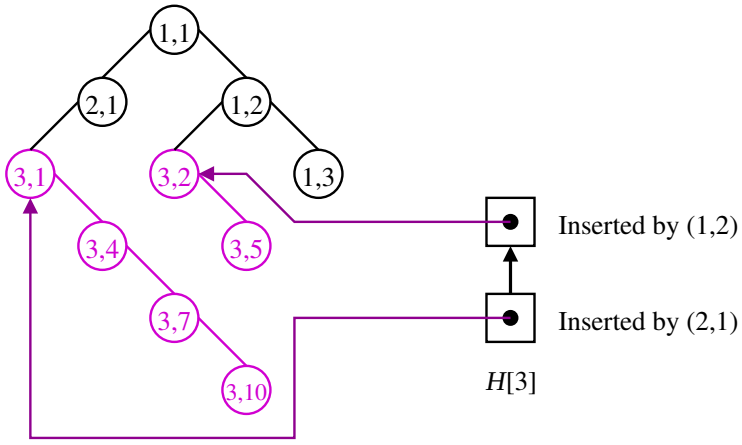
A straightforward approach is to make any traversal of T, and store the pairs in an external array. Then, sort the array, and print the sorted array. If n is the number of nodes in T, the sorting phase requires $O(n \log n)$ running time. But like *preorder* and *inorder*, we want *lexorder* to run in $O(n)$ time. Let A be the maximum value of a stored in T. Assuming that $A = O(n)$,[1] implement the following $O(n)$-time algorithm.

Create A linked lists $H[a]$ of pointers to T for $a = 1,2,3,\ldots,A$. Initialize $H[1]$ to the single-node list containing the pointer to the root node $(1,1)$. All other lists $H[a]$ are initially empty. For $a = 1,2,3,\ldots,A$ (in that order), repeat the procedure described in the next paragraph. See the figure on the next page.

Suppose that $H[a]$ contains k tree pointers. All these tree nodes store pairs (a,b) for the given a and with $b \leqslant a$ (equality holds only for $a = 1$). Moreover, the list $H[a]$ should be sorted with respect to the b values. Print the (a,b) pairs stored in the tree nodes pointed to by the pointers in $H[a]$ (from beginning to end). After each printing of the pair (a,b) stored at node v of T pointed to by the node u of $H[a]$, you need to do two additional things. First, if v has a left child, insert the pointer to this child at the beginning of $H[a+b]$. Second, check whether v has a right child. If not, delete u from $H[a]$. Otherwise, replace the pointer at u by the pointer to the right child of v. When you complete the pass through all the nodes of $H[a]$, you go back again to the beginning of $H[a]$, and make a second pass for doing the same things as in the first pass. Repeat until $H[a]$ becomes empty.

Free all the temporary memory associated with the linked lists $H[a]$ before the function returns.

[1] This is not the general case though. The maximum A may even be exponential in n. For example, consider the tree storing only the pairs (F_1,F_2), (F_3,F_2), (F_3,F_4), $\ldots$, (F_n,F_{n+1}), where F_i is the i-th Fibonacci number.



Lexicographic printing of nodes with a = 3

The *main*() function

- Initialize your pair tree T to a single-node tree containing the pair $(1,1)$.
- The user enters the numbers n_{ins} and n_{del} of insertions and deletions to be made on the tree.
- The user supplies n_{ins} pairs (a,b) for insertion to T. For simplicity, assume that $1 \leqslant a,b \leqslant 99$. Call *tinsert* on the supplied pairs.
- After all the insertions, print the preorder, inorder, and lexorder listings of the pairs stored in T.
- Randomly generate a pair (a,b) with $\gcd(a,b) = 1$, present in T. Print (a,b). Then, randomly generate a pair (a,b) with $\gcd(a,b) = 1$, not present in T. Print (a,b).
- The user supplies n_{del} pairs (a,b) for deletion from the tree. Call *tdelete* on the supplied pairs.
- After all these deletions, print the preorder, inorder, and lexorder listings of the pairs stored in T.

Sample output

```
nins = 10
ndel = 10

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+++ INSERTION PHASE
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

    (57,67): 11 nodes added
    (27,71):  9 nodes added
    (60,13):  8 nodes added
    (74,49): 24 nodes added
    ( 2,35): 17 nodes added
    (79,46):  6 nodes added
    (83,47):  7 nodes added
    (88,21): 10 nodes added
    (78,41): 11 nodes added
    (57,56): 33 nodes added

+++ Number of nodes = 137

+++ Preorder
    ( 1, 1) ( 2, 1) ( 3, 1) ( 4, 1) ( 4, 5) ( 4, 9) ( 4,13) ( 4,17) ( 4,21) (25,21) (46,21) (67,21)
    (88,21) ( 3, 4) ( 3, 7) (10, 7) (10,17) (27,17) (27,44) (27,71) ( 2, 3) ( 2, 5) ( 2, 7) ( 2, 9)
    ( 2,11) ( 2,13) ( 2,15) ( 2,17) ( 2,19) ( 2,21) ( 2,23) ( 2,25) ( 2,27) ( 2,29) ( 2,31) ( 2,33)
    ( 2,35) ( 1, 2) ( 3, 2) ( 3, 5) ( 8, 5) ( 8,13) (21,13) (34,13) (47,13) (60,13) ( 3, 8) ( 3,11)
    (14,11) (25,11) (36,11) (36,47) (83,47) ( 1, 3) ( 4, 3) ( 7, 3) ( 7,10) (17,10) (27,10) (37,10)
    (47,10) (57,10) (57,67) ( 1, 4) ( 5, 4) ( 9, 4) (13, 4) (17, 4) (21, 4) (25, 4) (29, 4) (33, 4)
    (37, 4) (37,41) (78,41) ( 1, 5) ( 1, 6) ( 7, 6) ( 7,13) (20,13) (33,13) (33,46) (79,46) ( 1, 7)
    ( 1, 8) ( 1, 9) ( 1,10) ( 1,11) ( 1,12) ( 1,13) ( 1,14) ( 1,15) ( 1,16) ( 1,17) ( 1,18) ( 1,19)
    ( 1,20) ( 1,21) ( 1,22) ( 1,23) ( 1,24) (25,24) (25,49) (74,49) ( 1,25) ( 1,26) ( 1,27) ( 1,28)
    ( 1,29) ( 1,30) ( 1,31) ( 1,32) ( 1,33) ( 1,34) ( 1,35) ( 1,36) ( 1,37) ( 1,38) ( 1,39) ( 1,40)
    ( 1,41) ( 1,42) ( 1,43) ( 1,44) ( 1,45) ( 1,46) ( 1,47) ( 1,48) ( 1,49) ( 1,50) ( 1,51) ( 1,52)
    ( 1,53) ( 1,54) ( 1,55) ( 1,56) (57,56)
```

```
+++ Inorder
    ( 4, 1) ( 4, 5) ( 4, 9) ( 4,13) ( 4,17) (88,21) (67,21) (46,21) (25,21) ( 4,21) ( 3, 1) ( 3, 4)
    (10, 7) (27,17) (27,44) (27,71) (10,17) ( 3, 7) ( 2, 1) ( 2, 3) ( 2, 5) ( 2, 7) ( 2, 9) ( 2,11)
    ( 2,13) ( 2,15) ( 2,17) ( 2,19) ( 2,21) ( 2,23) ( 2,25) ( 2,27) ( 2,29) ( 2,31) ( 2,33) ( 2,35)
    ( 1, 1) ( 3, 2) ( 8, 5) (60,13) (47,13) (34,13) (21,13) ( 8,13) ( 3, 5) ( 3, 8) (36,11) (83,47)
    (36,47) (25,11) (14,11) ( 3,11) ( 1, 2) ( 7, 3) (57,10) (57,67) (47,10) (37,10) (27,10) (17,10)
    ( 7,10) ( 4, 3) ( 1, 3) (37, 4) (78,41) (37,41) (33, 4) (29, 4) (25, 4) (21, 4) (17, 4) (13, 4)
    ( 9, 4) ( 5, 4) ( 1, 4) ( 1, 5) ( 7, 6) (33,13) (79,46) (33,46) (20,13) ( 7,13) ( 1, 6) ( 1, 7)
    ( 1, 8) ( 1, 9) ( 1,10) ( 1,11) ( 1,12) ( 1,13) ( 1,14) ( 1,15) ( 1,16) ( 1,17) ( 1,18) ( 1,19)
    ( 1,20) ( 1,21) ( 1,22) ( 1,23) (25,24) (74,49) (25,49) ( 1,24) ( 1,25) ( 1,26) ( 1,27) ( 1,28)
    ( 1,29) ( 1,30) ( 1,31) ( 1,32) ( 1,33) ( 1,34) ( 1,35) ( 1,36) ( 1,37) ( 1,38) ( 1,39) ( 1,40)
    ( 1,41) ( 1,42) ( 1,43) ( 1,44) ( 1,45) ( 1,46) ( 1,47) ( 1,48) ( 1,49) ( 1,50) ( 1,51) ( 1,52)
    ( 1,53) ( 1,54) ( 1,55) (57,56) ( 1,56)

+++ Lexical order
    ( 1, 1) ( 1, 2) ( 1, 3) ( 1, 4) ( 1, 5) ( 1, 6) ( 1, 7) ( 1, 8) ( 1, 9) ( 1,10) ( 1,11) ( 1,12)
    ( 1,13) ( 1,14) ( 1,15) ( 1,16) ( 1,17) ( 1,18) ( 1,19) ( 1,20) ( 1,21) ( 1,22) ( 1,23) ( 1,24)
    ( 1,25) ( 1,26) ( 1,27) ( 1,28) ( 1,29) ( 1,30) ( 1,31) ( 1,32) ( 1,33) ( 1,34) ( 1,35) ( 1,36)
    ( 1,37) ( 1,38) ( 1,39) ( 1,40) ( 1,41) ( 1,42) ( 1,43) ( 1,44) ( 1,45) ( 1,46) ( 1,47) ( 1,48)
    ( 1,49) ( 1,50) ( 1,51) ( 1,52) ( 1,53) ( 1,54) ( 1,55) ( 1,56) ( 2, 1) ( 2, 3) ( 2, 5) ( 2, 7)
    ( 2, 9) ( 2,11) ( 2,13) ( 2,15) ( 2,17) ( 2,19) ( 2,21) ( 2,23) ( 2,25) ( 2,27) ( 2,29) ( 2,31)
    ( 2,33) ( 2,35) ( 3, 1) ( 3, 2) ( 3, 4) ( 3, 5) ( 3, 7) ( 3, 8) ( 3,11) ( 4, 1) ( 4, 3) ( 4, 5)
    ( 4, 9) ( 4,13) ( 4,17) ( 4,21) ( 5, 4) ( 7, 3) ( 7, 6) ( 7,10) ( 7,13) ( 8, 5) ( 8,13) ( 9, 4)
    (10, 7) (10,17) (13, 4) (14,11) (17, 4) (17,10) (20,13) (21, 4) (21,13) (25, 4) (25,11) (25,21)
    (25,24) (25,49) (27,10) (27,17) (27,44) (27,71) (29, 4) (33, 4) (33,13) (33,46) (34,13) (36,11)
    (36,47) (37, 4) (37,10) (37,41) (46,21) (47,10) (47,13) (57,10) (57,56) (57,67) (60,13) (67,21)
    (74,49) (78,41) (79,46) (83,47) (88,21)

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+++ SEARCH PHASE
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

    ( 8,13): Search successful
    (78,47): Search failed

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+++ DELETION PHASE
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

    ( 5, 4): 11 nodes deleted
    ( 1,13): 48 nodes deleted
    (33,13):  3 nodes deleted
    (60,13):  1 nodes deleted
    ( 1, 6): 10 nodes deleted
    ( 4,21):  5 nodes deleted
    ( 3, 4):  7 nodes deleted
    ( 3,11):  6 nodes deleted
    (37,10):  4 nodes deleted
    (27,10):  1 nodes deleted

+++ Number of nodes = 41

+++ Preorder
    ( 1, 1) ( 2, 1) ( 3, 1) ( 4, 1) ( 4, 5) ( 4, 9) ( 4,13) ( 4,17) ( 2, 3) ( 2, 5) ( 2, 7) ( 2, 9)
    ( 2,11) ( 2,13) ( 2,15) ( 2,17) ( 2,19) ( 2,21) ( 2,23) ( 2,25) ( 2,27) ( 2,29) ( 2,31) ( 2,33)
    ( 2,35) ( 1, 2) ( 3, 2) ( 3, 5) ( 8, 5) ( 8,13) (21,13) (34,13) (47,13) ( 3, 8) ( 1, 3) ( 4, 3)
    ( 7, 3) ( 7,10) (17,10) ( 1, 4) ( 1, 5)

+++ Inorder
    ( 4, 1) ( 4, 5) ( 4, 9) ( 4,13) ( 4,17) ( 3, 1) ( 2, 1) ( 2, 3) ( 2, 5) ( 2, 7) ( 2, 9) ( 2,11)
    ( 2,13) ( 2,15) ( 2,17) ( 2,19) ( 2,21) ( 2,23) ( 2,25) ( 2,27) ( 2,29) ( 2,31) ( 2,33) ( 2,35)
    ( 1, 1) ( 3, 2) ( 8, 5) (47,13) (34,13) (21,13) ( 8,13) ( 3, 5) ( 3, 8) ( 1, 2) ( 7, 3) (17,10)
    ( 7,10) ( 4, 3) ( 1, 3) ( 1, 4) ( 1, 5)

+++ Lexical order
    ( 1, 1) ( 1, 2) ( 1, 3) ( 1, 4) ( 1, 5) ( 2, 1) ( 2, 3) ( 2, 5) ( 2, 7) ( 2, 9) ( 2,11) ( 2,13)
    ( 2,15) ( 2,17) ( 2,19) ( 2,21) ( 2,23) ( 2,25) ( 2,27) ( 2,29) ( 2,31) ( 2,33) ( 2,35) ( 3, 1)
    ( 3, 2) ( 3, 5) ( 3, 8) ( 4, 1) ( 4, 3) ( 4, 5) ( 4, 9) ( 4,13) ( 4,17) ( 7, 3) ( 7,10) ( 8, 5)
    ( 8,13) (17,10) (21,13) (34,13) (47,13)
```

Print the number of nodes created or deleted by a *tinsert* or *tdelete* operation. These functions should return these counts. Maintain the size n of T externally.

Submit a single C/C++ source file. Do not use global/static variables.