Please follow the instructions provided below to submit your assignment.

Worth: 10% **Due:** By 11:59 pm on Friday Oct 20

- You must submit your assignment as a single PDF file through the MarkUs system.

  https://markus.teach.cs.toronto.edu/csc263-2017-09/

  The filename must be the assignment's name followed by "sol", e.g. your submission for the assignment "A2" must be "A2sol.pdf". Any group of two students would submit a single solution through Markus. Your PDF file must contain both team member's full names.
- Make sure you read and understand "POLICY REGARDING PLAGIARISM AND ACADEMIC OFFENSE" provided in the course information sheet.
- The PDF file that you submit must be clearly legible. To this end, we encourage you to learn and use the LaTex typesetting system, which is designed to produce high-quality documents that contain mathematical notation. You can find a latex template in Piazza under resources. You can use other typesetting systems if you prefer. Handwritten documents are acceptable but not recommended. The submitted documents with low quality that are not clearly legible, will not be marked.
- You may not include extra descriptions in your provided solution. The maximum space limit for each question is 2 pages. (using a reasonable font size and page margin)
- For any question, you may use data structures and algorithms previously described in class, or in prerequisites of this course, without describing them. You may also use any result that we covered in class, or is in the assigned sections of the official course textbook, by referring to it.
- Unless we explicitly state otherwise, you should justify your answers. Your paper will be marked based on the correctness and completeness of your answers, and the clarity, precision, and conciseness of your presentation.
- For describing algorithms, you may not use a specific programming language. Describe your algorithms clearly and precisely in plain English or write pseudo-code.

1. A binary search tree is *weight balanced* if for each node in the tree, the number of nodes in its two subtrees differ by at most 1. Recall that for an AVL tree, it is the height of the two subtrees that must differ by at most one.

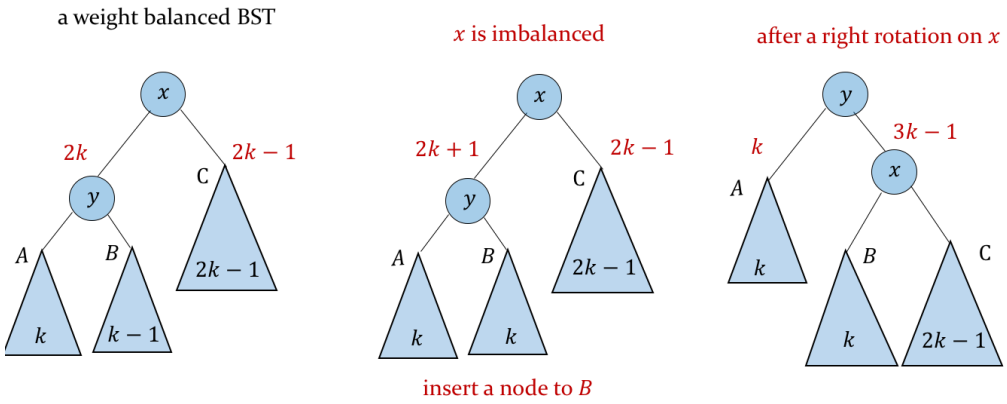
   (a) (10 MARK) Give an algorithm to build a weight balanced BST given a sorted array of n elements. Your algorithm should run in $O(n)$ time. Analyze the time complexity of your algorithm.

   - Algorithm:
     This definition for a weight balanced BST of size n implies that the size of the subtrees should be $n/2, n/2-1$ if n is even and $(n-1)/2, (n-1)/2$ if n is odd . It means that the root of a weight balanced tree stores the median of the elements. So, the idea is to insert the median of the elements recursively to the left and right tree.
     The algorithm computes the median index of the array and inserts its element as the root of the tree. The left and right subtrees are computed recursively for the left half of the array and the right half of the array. The procedure terminates when the size of the array becomes zero.
   - Time complexity:
     Let $T(n)$ be the worst case time complexity of the recursive algorithm
     - Finding the median of an array and insert to the root: $O(1)$.
     - Recursion: $T(n) = 2T(n/2) + O(1)$
     - Using master theorem: $T(n) = O(n)$

   (b) (10 MARK) Prove that the height of a weight balanced BST is $O(logn)$.
   Let $h(n)$ be the height of a weight balanced BST with n elements. We use induction to prove that $h(n) \leq log(n)$.
   - Base case: $h(1) = 0 \leq \log(1)$
   - We assume $h(k) \leq \log k$ for $1 \leq k \leq n-1$ and we prove $h(n) \leq \log(n)$
     Considering two cases for the size the tree:
     - n is even: the subtrees have the size of $n/2, n/2-1$.
       $h(n) = \max\{h(n/2), h(n/2-1)\}+1 \leq \max\{\log(n/2), \log(n/2-1)\}+1 = \log(n/2)+1 = \log n$
     - n is odd: the subtrees have the size of $(n-1)/2, (n-1)/2$.
       $h(n) = \max\{h((n-1)/2), h((n-1)/2)\} + 1 \leq \log((n-1)/2) + 1 = \log(n-1)$

   In both cases, $h(n) \leq logn$.

   (c) (5 MARK) Does the same rebalance operation for AVL tree insert work for inserting to weight balanced BST? Justify your answer by a proof or a counter example.
   A counter example exists.
   - The initial tree must be a weight balances BST as stated in the question (insert to a weight balanced BST).
   - After insertion it must be imbalanced. (After insertion the rotation is performed on the lowest node that violates the weight-balanced BST. (single or double rotation)
   - Rotation(s) must be correct and final result imbalanced.



2. Bob has m dollars and wants to buy two items with an exact total cost of m. Assume that you are given a binary search tree that stores the price of n available items and no two items have the same price.

   (a) (15 MARK) Write an $O(n)$ algorithm to find two items with total price of m.

   The idea is to use two pointers (forward and backward), the forward pointer traverses the tree by an in-order walk starting from the element with the minimum key and the backward pointer traverses the tree by reverse in-order walk starting from the element with maximum key.

   - If the tree has less than 2 nodes, it returns "no pair" and exits.
   - The algorithm finds the minimum and maximum keys in the tree and assigns the minimum to backward and the maximum to the forward.
   - While forward $\neq$ backward, the algorithm computes sum of the keys in the forward and backward elements.
     - if $sum == m$, then it returns the forward and backward elements and exists.
     - if $sum < m$, then forward would be updated by the forward node's successor.
     - if $sum > m$, then backward would be updated by the backward node's predecessor.
   - Algorithm returns "no pair" and exits.

   Another approach which needs an extra $O(n)$ space, is to traverse the tree by an in-order walk and copy all the elements in an array. Then use a similar approach.

   (b) (5 MARK) Analyze the time complexity of your algorithm.

   Tow pointers, backward and forward are updated at most $n-1$ times and each time, it calls the procedure of successor or predecessor. We proved in exercise 12.2-7 calling tree-minimum and then making $n-1$ calls to tree-successor would run in $\Theta(n)$. By symmetry, we can prove that calling tree-maximum and then making $n-1$ calls to tree-predecessor would run in $\Theta(n)$ as well. The other steps of the algorithm run in a constant time. So, the algorithm in total would run in $O(n)$.

3. Consider the following opertation in a binary tree that stores integer keys:

   - CLOSEST-PAIR(r): returns the minimum difference of the keys among all pairs in the subtree rooted at r. In other words, Closest-pair(r) returns the minimum value of $|x.key - y.key|$ among all elements x, y rooted at r.

   We want to is augment AVL tree to support this operation in $O(1)$ time such that the time complexity of *Insert* and *Delete* remains $O(\log n)$.

   (a) (6 MARK) What extra information do we need to store at each node?

   - r.max: The maximum value of the subtree rooted at r.
   - r.min: The minimum value of the subtree rooted at r .
   - r.closest-pair: The minimum difference of the subtree rooted at r.

   (b) (8 MARK) Explain how to perform CLOSEST-PAIR in $O(1)$?

   (c) (16 MARK) Explain why Insert and Delete can still be performed in $O(\log n)$ time?

   CLOSEST-PAIR(r)
   . **return** r.closest-pair

   CLOSEST-PAIR(r) simply returns r.closest-pair, so it can be computed in O(1) since its attributes are accessible in a constant time. That is possible when we augment each node in the tree with the required attributes.

   We show that r.min, r.max and r.closest pair can be updated by the attributes of r, r.left and r.right.

   r.min = min {
   r.key
   r.left.min if r.left exists
   r.right.min if r.right exists
   }

   r.max = max {
   r.key
   r.left.max if r.left is not null.
   r.right.max if r.right is not null.
   }

   If the left and right subtrees are empty, then we define the closest pair to be infinity. (It is also can be defined null and then a few other cases should be considered.)

r.closest-pair = min {

r.left.closest-pair,	if r.left is not null
r.right.closest-pair,	if r.right is not null
r.key - r.left.max,	if r.left is not null
r.right.min - r.key	if r.right is not null
$+\infty$	if both r.right and r.left are null

}

For any node that is inserted or deleted, we use the AVL-insert or AVL-delete algorithms learned in the lecture. In insert, the attributes of inserted node plus the attributes of a constant number of nodes involved in the single or double rotation get changed and in delete the attributes of deleted node plus the attributes of the node that is moved to its place plus the attributes of a constant number of nodes involved in the single or double rotation get changed. When attributes of a node is changed in an AVL-tree, only its parents' and ancestors' attributes need to be updated ($O(\log n)$ nodes). Since, in total, a constant number of nodes (i.e. $O(1)$) get changed and since the update of attributes of a single node like r happens in $O(1)$ using the above formulas, based on theorem 14.1 in the textbook, the total time complexity of insert and delete would be $O(\log n)$.

4. A *cryptographic hash function* is a particular class of hash functions that have special properties which make them suitable for use in cryptography. The ideal cryptographic hash function has five main properties:

   1) It is deterministic so the same message always results in the same hash.
   2) It is quick to compute the hash value for any given message.
   3) It is infeasible to generate a message from its hash value except by trying all possible messages.
   4) A small change to a message should change the hash value so extensively that the new hash value appears. uncorrelated with the old hash value
   5) It is infeasible to find two different messages with the same hash value

   In this question, you will see the algorithms for two simple string hash functions. For both algorithms, the input is a string of ASCII characters and the output is a hash code.

   - **Algorithm 1**

     This algorithm is not the best possible algorithm, but it has the merit of extreme simplicity.

     ```
     1: function HASH(S)
     2:     hash = 0
     3:     for each characters c in S do
     4:         hash += ASCII(c)
     5:     end for
     6:     return hash
     7: end function
     ```

   - **Algorithm 2**

     This algorithm is one of the best algorithms known for string hash.

```
1: function HASH(S)
2:     hash = 5381
3:     for each characters c in S do
4:         hash = hash * 33+ASCII(c)
5:     end for
6:     return hash
7: end function
```

(a) (7 MARK) What is the hash code of the string "abcd" using Algorithm 1 and Algorithm 2?

- Using algorithm 1,
  Hash("*abcd*") = 97 + 98 + 99 + 100 = 394.

- Using algorithm 2,
  Hash("*a*") = (5381 * 33) + 97 = 177670
  Hash("*ab*") = (177670 * 33) + 98 = 5863208
  Hash("*abc*") = (5863208 * 33) + 99 = 193485963
  Hash("*abcd*") = (193485963 * 33) + 100 = 6385036879

(b) (5 MARK) For each 5 properties of cryptographic hash functions mentioned above, discuss if Algorithm 1 has the property. Briefly justify your answer for each or give a counter example.

- It has Property 1 because ASCII codes are constant and the algorithm's result for an input would be always the same.
- This is considered to be fast since it simply just computes the sum of all the character's ASCII values in $O(n)$.
- Algorithm 1 does not hold Property 3 because if we are given a hash value, it is easy to find a combination of values in ASCII range, whose total sum is equal to the given hash value. We can chose semi-random values for the first $n-1$ chars and adjust the last char accordingly to have the required total sum. For example, if we are given 197 as the hash value, two combinations can be "*ad*" and "*bc*."
- It does not have Property 4 because a small change in message does not produce a disproportionate change in the hash value. A counter example would be changing "abcd" to "bbcd", the hash value 394 would become 395.
- It does not have Property 5 because the hash value is computed from a simple sum, you can easily find two strings with the same hash values for example by adding 1 to the first char and subtracting 1 from the last one as shown in "*ad*" and "*bc*").

(c) (5 MARK) Why do you think Algorithm 2 is a better algorithm? (Specify at least one property in which you think algorithm 2 performs better than Algorithm 1. Justify your answer.)

Properties 3, 4 and 5 are satisfied slightly better using Algorithm 2 because it has a more complex procedure for generating the hash which makes it harder to reverse the hash function or find collisions. Also it provides better propagation of changes, which means by changing a single char in a string the hash value would be much more different compared to Algorithm 1.

(d) (8 MARK) Name two well-known cryptographic hash functions and specify the hash code of the string "abcd" using each of them. (You do not need to describe the algorithms and the computation process. You might need a quick Google search or running a command line program.)

MD5, SHA1, SHA256, ...
MD5: e2fc714c4727ee9395f324cd2e7f331f
SHA1: 81fe8bfe87576c3ecb22426f8e57847382917acf
SHA256: 88d4266fd4e6338d13b845fcf289579d209c897823b9217da3e161936f031589