Computer Algorithm

CIE-3090

Fall 2017
Term Project

IELTS Registration

Group: CSE-15-1

Student Names	Student ID
Sergey Kim	U1510382
Andrey Kim	U1510307
Sergey Bya	U1510347

Abstract:

To begin with, many people nowadays learn English language for several reasons. They migrate to foreign countries, enter international universities, seek for a job that requires knowledge of English language and so on. To perform such activities, certificate is required to serve as a proof of competent language knowledge. Furthermore, there are plenty of them but the most popular now are IELTS and TOEFL (Testing Of English as Foreign Language).

IELTS acronym stands for International English Language Testing System. It is held in more than 30 countries across the world. There are two modules: academic, which is for university application, and general that is required for migration to English-speaking country.

To receive IELTS certificate, an examinee needs to take four sections of exam itself (Speaking, Reading, Listening, Writing). Each particular section is banded 0 to 9 according to specific criteria, i.e. for Speaking exam they are: speech fluency, academic vocabulary, grammar. Ordinary, British Council holds speaking test one week before other exams. The next 3 sections (Listening, Writing, Reading) take place in single day and are estimated the same way including specific criteria. Then, the whole examination sheet is properly checked and applicant's grades are calculated. Overall IELTS band is represented as average among 4 sections mentioned above.

The procedure starts with the process of examinee registering on an appropriate and available exam date. From the applicant's perspective, sequence of operations is impressed as:

1. Choose module.
2. Look at available dates.
3. Choose appropriate date.
4. Input personal data.
5. Pay exam fee.
6. Wait for a confirmation.

Meanwhile, authorities of British Council that creates examination dates do the following:

1. Create date (usually 2 days a month for academic and 2 for general).
2. Confirm applicants.
3. Grade sections.
4. Calculate overall mark.

Deletion of dates.

There is a probability, that deletion of date is required. For example, in some period of an academic year, let's say in June, there are much more applicants due to the reason that June is the last month to apply documents to university. In this case, many applicants procrastinates decision to register earlier, because they wish to prepare better. The number of vacant places is limited and authorities of British Council hold extra dates to give them opportunities to succeed. Somehow, miscalculation occurred or applicants changed their mind and refused implying that administrators deal with unnecessary dates (junk keys in a tree). To release space, trash must be deleted.

Design:

AVL Tree is a self-balancing binary search tree where the difference between heights of left and right subtrees cannot be more than one for all nodes.

Analyzing key features of project, we decided to use AVL Tree due to following reasons:

1. The most frequent operation is searching an examinee via ID and exam date.
2. Assignment of grades to each person.
3. Creation of examination dates.

The whole idea is that examination dates in major cases come in ascending order. There are no duplicate dates and each next date is mathematically greater than previous. Actually, in Uzbekistan IELTS is held 4 times a month. Several months are available for registration in advance.

Now, assume that each date is greater than previous, i.e.:

December dates:

Academic module: 7th December 2017, 24th December 2017.

General module: 4th December 2017, 21st December 2017.

If exam manager creates first date as 4th of December, then 7th, then 21st and 24th respectively, usual Binary-search Tree will be skewed in right side:

Root = 4;

Root.right = 7;

Root.right.right = 21;

Root.right.right.right = 24;

Time complexity of such data structure will not satisfy us, because it depends on height of the tree. Nevertheless, using rotations of AVL Tree during insertion, dates are sorted in proper order. While search a required date the result is returned in O(logn) time, which is a great outcome.

So, to give more precise information, each key in AVL Tree contains an array of examinees. There are 255 people at most in 1 exam date. This array is design in terms of making its' index as student's ID, which allows to access a specific person in constant time. Combining the time complexity of tree itself and nested array inside, to get an examinee takes O(logn) * O(1) operations. If we used a different data structure, say Queue or Stack, we could access data in O(1) time. Such enhancement is possible only if registration is done on the single available date. People do not have an opportunity to choose wished day in advance. The key that represents date is pushed onto Stack or Queue. Then, each key has an attribute interpreted as array of students. Accessing particular person takes time to pop() and to specify his ID that is index in array: O(1) * O(1).

Unfortunately, it is a poor idea because there are too many applicants for a single date. People will not be able to plan in a long run leading to inconveniency. Moreover, there are very usual situations, where more than 4 exams a month are organized additionally:

1. There are too many applicants, who wish to take exam in particular month. A live example is a June month, mentioned above.
2. Alternatively, in some cases absence of examinees leads to cancelation of common dates.

   It is impossible to predict these human factors, even in our university there was a similar situation, where British Council opened 2 exams separately for a single establishment. Thus, dealing with unstable scenarios, we need a stable algorithm that works in unchanged time despite any cases occurred.

We will describe 3 major functions: Insertion, Deletion, Access.

Before start, look at design of tree's node, that contains several attributes:

```
Node {
Key                 //date of exam
Height = 1          //height of node which is initially 1 for each created node
Left                //left child of node
Right               //right child of node
Examinees[255]      //array of 255 examinees
};
```

```
AVL_Tree {
Root                                    //root of the tree
Height(node)                            //height procedure
{
    if node is NULL
        return 0;
    else return node.hegiht;
}
getBalance(node)                        //procedure that calculates balance
{
    if node is NULL
        return 0;
    else return height(node.left) – height(node.right);
}
};
```

Implementation:

```
insert (node, key)
{
        if node is NULL then
             return newNode(key);

        if key < node.key then
             return insert(node.left, key);

        if key > node.key then
             return insert(node.right, key);

        node.height =
        1 + max(height(node.left),
        height(node.right));

        balance = getBalance(node);

        if balance > 1 and key < node.left.key then
             return rotateRight(node);

        if balance < -1 and key > node.right.key then
             return rotateLeft(node);

        if balance > 1 and key > node.left.key then
             node.left = rotateLeft(node.left);
             return rotateRight(node);

        if balance < -1 and key < node.right.key then
             node.right = rotateRight(node.right);
             return rotateLeft(node);

        return node;
}
```

Explanation:

The procedures height and getBalance are executed in constant time. They only return some values.

As in usual BST insertion is made in a sense of placing minimum key at left and maximum at right.

1. Check whether there is an allocated space or not. If tree is empty, insert node as root. If not, check key to be smaller or greater than corresponding node. In case inserted key is smaller, make it a left child of node or make is right child if key is greater.
2. Height of each node at initialization is 1. That is, if there are N levels in a tree, height of root is N and height of bottom level nodes is 1. AVL tree keeps itself balanced by updating the height of node at each insertion step. Height of such node is 1 + the height of one of its' child that is more than child's sibling.

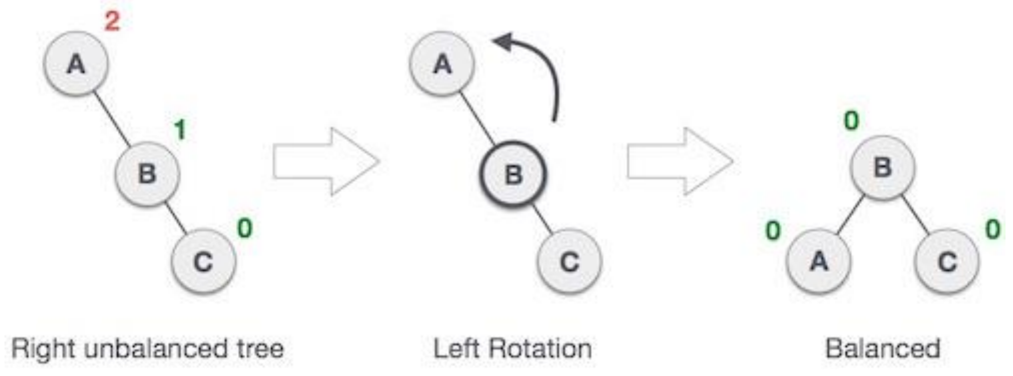
3. Balance is treated as difference between node’s left and right children. If balance becomes more than modulus (1), then AVL tree property is violated. There are 4 possible scenarios of violation, which are maintained by rotations:

```
rotateLeft(node)
{

      entry_right = node.right;

      entry_right_left = entry_right.left;

      entry_right.left = node;

      entry.right = entry_right_left;

      node.height = 1 + max(height(node.left), height(node.right));



      entry_right.height = 1 + max(height(entry_right.left),
      height(entry_right.right));

      return entry_right;

  }

  rotateRight()

  {
      entry_left = node.left;
      entry_left_right = entry_left.right;

      entry_left.right = node;
      entry.left = entry_left_right;

      node.height = 1 + max(height(node.left), height(node.right));


      entry_left.height = 1 + max(height(entry_left.left),
      height(entry_left.right));

      return entry_left;
  }
```

Rotations serve as balancing tool for maintaining height of given property. To explain rotations, here is the diagram:

Left Rotation

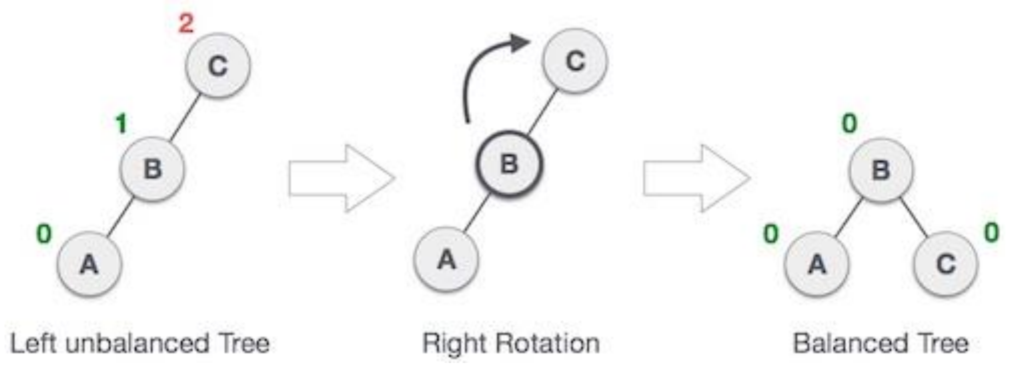
If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation −



In our example, node **A** has become unbalanced as a node is inserted in the right subtree of A's right subtree. We perform the left rotation by making **A** the left-subtree of B.

Right Rotation

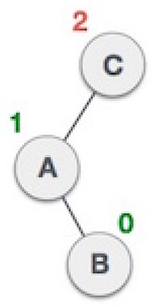
AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.

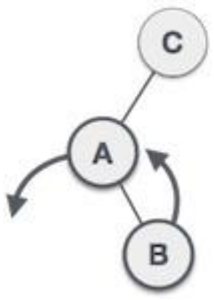
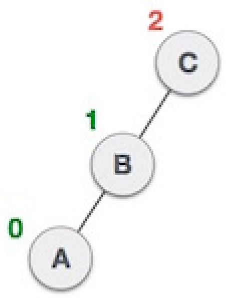
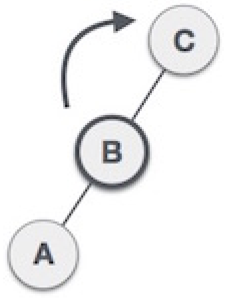
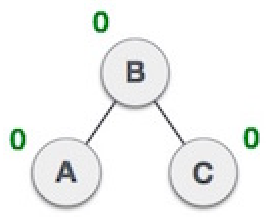


As depicted, the unbalanced node becomes the right child of its left child by performing a right rotation.

Left-Right Rotation

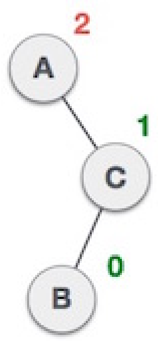
Double rotations are slightly complex version of already explained versions of rotations. To understand them better, we should take note of each action performed while rotation. Let's first check how to perform Left-Right rotation. A left-right rotation is a combination of left rotation followed by right rotation.

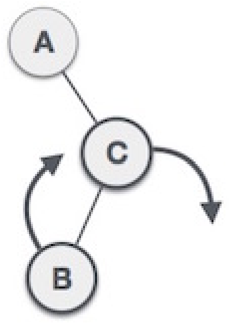
State	Action
	A node has been inserted into the right subtree of the left subtree. This makes **C** an unbalanced node. These scenarios cause AVL tree to perform left-right rotation.

	We first perform the left rotation on the left subtree of **C**. This makes **A**, the left subtree of **B**.
	Node **C** is still unbalanced, however now, it is because of the left-subtree of the left-subtree.
	We shall now right-rotate the tree, making **B** the new root node of this subtree. **C** now becomes the right subtree of its own left subtree.
	The tree is now balanced.

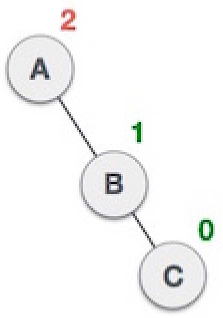
Right-Left Rotation

The second type of double rotation is Right-Left Rotation. It is a combination of right rotation followed by left rotation.

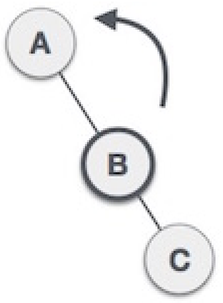
State	Action
	A node has been inserted into the left subtree of the right subtree. This makes **A**, an unbalanced node with balance factor 2.



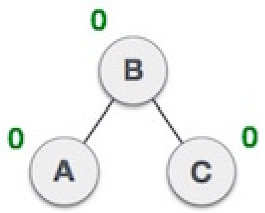
First, we perform the right rotation along **C** node, making **C** the right subtree of its own left subtree **B**. Now, **B** becomes the right subtree of **A**.



Node **A** is still unbalanced because of the right subtree of its right subtree and requires a left rotation.



A left rotation is performed by making **B** the new root node of the subtree. **A** becomes the left subtree of its right subtree **B**.



The tree is now balanced.

4. Finally, node is returned and procedure terminates. The key moment of insertion algorithm locates in maintaining balance changing height of nodes simultaneously. Rotations play the most essential role making impact on sorting dates via insertion.

Mapping ID as Index of array:

To retrieve each student in constant time, we decided to interpret ID as index of array. More precisely, we have an array, that is:

examinees[255]

actually number of applicants can be any, and indexing starts from 0 to 254. People do not care which ID they have, whether it is a generated combination of digits or characters or just a number 0, 1, 2, … , 254. However, if we wished, we could have encrypted array index to be a more complex ID using simple mathematical operations. Of course, it occupies performance because of additional operations. In summary, an examinee is accessed in O(1) time, equivalently to: examinees[wished ID].

Access through date:

To retrieve a wished date, of course we need to search it. Assume, Tree has some height as well as every node in it. As the key is found on some node k, which height is h, the complexity is O(h). The worst case is the height of the tree, when the key is located on the last level of the tree. However, as AVL Tree is a tree with minimum height, that is equal to log(n), operation requires O(logn) complexity to be done, due to tree structure and recursive calls. It is done by the following method:

```
Algorithm access(node, key)
{
   if node is NULL then
      terminate;
   if key < node.key then
      node = access(node.left, key);
   if key > node.key then
      node = access(node.right, key);
   return node;
}
```

Access explanation:

To sum up, we have individual unique dates that are keys. Manager of exam assigns dates that are stored into the AVL tree container. After each insertion, tree is balanced, keys are sorted and examinees allocated to each date as well.

1. If the corresponding node is not allocated, that means there is no such date in a tree.
2. Each time control flow travels level down, number of operations are twice less, which means logn complexity: T(n / 2).
3. As AVL tree is sorted during insertion step, we search for a key travelling left or right depending on condition.
4. Finally, a date is accessed in O(logn) complexity, which is good for retrieving student.

```
Deletion algorithm:

erase(node, key)

{

    entry = access(node, key);


    if entry.left is NULL or entry.right is NULL then

         temp = entry;

         if entry.left is NULL then

              temp = entry.right;

         else temp = entry.left;


         if temp is NULL then

              temp = node;

              node = NULL;

         else node = temp;

              delete temp;

    else temp = getMinimumValue(node.right);

         node.key = temp.key;

         node.right = erase(node.right, temp.key);

    Update height of node;

    Check balance;

    Rotate if necessary;

    Return node;

}
```

Delete explanation:

First and foremost, algorithm search for a node that have corresponding key. As in insert and access, control flow goes deeper according to the condition, whether key is smaller of greater.

When the key is found:

1. Checks whether one of its children is NULL.
2. If so, check whether left child is NULL. If so, create a temporary variable that contains right child of found node, a.k.a. left child's sibling.

3. Else, right child of corresponding node is NULL and temp equals left child.
4. If temp is NULL (left of right child of desired node), temp becomes node itself, and node becomes NULL.
5. Else, node equals temp (left or right child that is not NULL).
6. Delete temp that is node.
7. If there are two children of corresponding node, then find the inorder successor, give contents of node's children to successor and delete this node.
8. If some unbalancing is found, rotate as in insert procedure to balance the tree.

Time Complexity:

1. Insert:
   Looking at procedure insert(node, key) there are operations and their repetitions to add 1 element to the AVL tree. Each operation takes some constant time c. Actually, for any operation in function exact time is different and is c_i. However, to simplify assumption, consider each time as c, because we need only complexity, not actual time.
   Insert(node, key) contains no loops but recursive calls and branch operations (if/else). For a single call, there are 3 conditions at the beginning, from which only 1 occurs at a time. Then, there are 2 recursive calls, which repeat [T/2] time to execute, since it is a binary tree. Consequently, the next 4 cases that describe unbalanced scenario also execute at constant time, because only 1 condition of specific property violation can occur. Height and Balance calculation are also constant time functions. Equally important, rotations contain no loops and recursive calls, only assignment and other constant operators, which means, they are O(1).
   Finally, to simulate insertion, we need:

   Time for access + Time to calculate height of each node on return + Time to calculate balance of each node + Time for rotations, that is:

   $T(n) = \log(n) + c_0 * \log(n) + c_1 * \log(n) + c_2 * \log(n)$, where
   c_0 – time to calculate height of 1 node,
   c_1 – time to calculate balance of 1 node,
   c_2 – time to perform single rotation. Rotation has no loops and recursive calls, which means it runs in some constant time. We omit it and express as c_2.
   So, the running time of tree with n nodes is: $T(n) = \log(n)$.

2. Access operation is the first branches of insertion that is a simple recursive search of a required node. To execute, algorithm requires O(logn) time, because maximum height is log(n).
3. Erase procedure is the same kind of approach as access and insert. Constant operations only different in these 3 functions but we can easily omit them. Precisely, erase includes access, height and balance calculation, rotations, if needed and some operations, depending on conditions of 3 occurrences:
   1. Node to be erased has no children: just erase corresponding node. O(1).
   2. Node to be erased has 1 child: child is putted on the place of corresponding node and node is erased. O(1).

3. Node to be erased has 2 children: inorder traversal on corresponding node, which runs in O(h), where h is a height of this node and recursive calls on the right child of this node, which runs in height of its’ right child. O(logn) + O(logn – 1) = O(logn).

Finally, worst case of delete includes (i.e. delete root): access with max height, updating all nodes, rotating tree h times, performing height and balance updates h times. Overall, there is no a factor that is more than log(n), consequently, erase(root, key) = W(n) = log(n).