CSCI-GA.1170-001/002 Fundamental Algorithms October 20, 2019

Solutions to Problem 1 of Homework 6 (15 Points)

Name: jingshuai jiang (jj2903) *Due: 5 pm on Thursday, October 17*

Collaborators: NetID1, NetID2

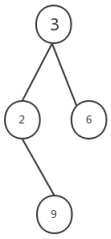
Let us recall the definition of *binary search tree property*: Let x be a node in a binary search tree. If y is a node in the left subtree of x, then $y.key \leq x.key$. If y is a node in the right subtree of x, then $y.key \geq x.key$

Any binary tree that satisfies this property is a binary search tree. The goal of this question is to create an algorithm to check if a given binary tree is also a binary search tree.

(a) (2 points) Consider the following algorithm: For each node, check if its left child is lesser than its key and its right child is greater than its key. If yes, pass the recursion onto the left and right child. This pseudocode is as follows:

```
1 IsBST-A(x)
2     if (x = NIL)
3         return True
4     if (x.left! = nil  &  x.left.key > x.key)
5         return False
6     if (x.right! = nil  &  x.right.key < x.key)
7         return False
8     if (IsBST-A(x.left)=False Or IsBST-A(x.right)=False)
9         return False
10    return True
```

Show that this algorithm is wrong by constructing an example of a binary tree which is not a binary search tree but for which the algorithm returns true.



Solution: this is a wrong tree □

(b) (3 points) The goal of this question is to modify IsBST-A to make it work. To achieve this, you may use two helper helper functions FindMin(v), FindMax(v) that returns the

minimum and maximum value from a non-empty sub-tree rooted at v. These are trivial $O(n)$ algorithms that take as input the node of a binary tree and returns the minimum and maximum values respectively in the subtree rooted at that node. Construct this algorithm IsBST-B by minimally modifying IsBST-A. Argue the correctness of your algorithm. (**Hint**: You have to change only two lines in the code for IsBST-A.)

Solution:

```
1 IsBST-A(x)
2     if (x = NIL)
3         return True
4     if (x.left! = nil  &  FindMax(x.left) > x.key)
5         return False
6     if (x.right! = nil  &  FindMin(x.right) < x.key)
7         return False
8     if (IsBST-A(x.left)=False Or IsBST-A(x.right)=False)
9         return False
10    return True
```

□

(c) (3 points) What is the recurrence relation for the runtime of the above algorithm? Solve the same to give the worst-case runtime of your algorithm. (**Hint**: Let n_1, n_2 be the number of nodes in the left and right subtrees. Then, $n_1 + n_2 = n - 1$.)

Solution:

$$T(n) = T(k) + T(n-k-1) + O(n-1) = \Theta(nlogn)$$

□

(d) (4 points) Modify IsBST-B to create a linear time algorithm IsBST-C. This algorithm takes as input x indicating the subtree rooted at node x. It should return three values: the minimum element, the maximum element in the subtree rooted at x, and a boolean value indicating if the subtree is a BST or not. Argue the correctness of your algorithm.

Solution:

```
1 IsBST-C(x)
2     if (x = NULL)
3         return NULL NULL True
4     if (x.left! = NULL)
5         leftmin, leftmax, leftTruth = IsBST-C(x.left)
6     else leftmin, leftmax, leftTruth =x.value,x.value,True
7     if (x.right! = NULL)
8         rightmin, rightmax, rightTruth = IsBST-C(x.right)
9     else rightmin, rightmax, rightTruth =x.value,x.value,True
```

```
10    x.max = Max(leftmax,rightmax,x.value)
11    x.min = Min(leftmin,rightmin,x.value)
12    if (leftmax>x.value or rightmin<x.value or leftTruth == false or rightTruth ==false)
13          return x.min, x.max, FALSE
14    else
15          return x.min, x.max, TRUE
```

We carefully examine each node by comparing whether its value is greater than all the values of its leftchild, whether its value is less than all the values of its rightchild and whether its subtree is binary search tree to decide whether it is a binary search tree.

We compare the minvalue of its leftchild, the minvalue of its rightchild and the value of itself to determine the minimum value.

We compare the maxvalue of its leftchild, the maxvalue of its rightchild and the value of itself to determine the maximum value.

In this process, each node has been checked for only one time. So it is a linear time algorithm.

□

(e) (3 points) What is the recurrence relation for the runtime of the algorithm? Solve the same to show that the worst-case runtime is $O(n)$. (**Hint**: Let n_1, n_2 be the number of nodes in the left and right subtrees. Then, $n_1 + n_2 = n - 1$.)

Solution:

$$T(n) = T(k) + T(n-k-1) + O(1)$$

If the tree always does not have its leftchild or it always does not have its rightchild,then we get the worst case,which is

$$T(n) = T(0) + T(n-1) + O(1) = T(n-1) + O(1) = O(n)$$

□

CSCI-GA.1170-001/002 Fundamental Algorithms October 20, 2019

Solutions to Problem 2 of Homework 6 (18 points)

Name: jingshuai jiang (jj2903) *Due: 5 pm on Thursday, October 17*

Collaborators: NetID1, NetID2

You are given various inputs corresponding to certain walk(s) of a binary tree T. The goal is to recover T (uniquely) from these walks (i.e, provide a pseudocode, argue its correctness and runtime using a recurrence relation) or provide counterexamples to show that recovering such a unique tree is not possible. The counterexample should contain the minimum number of nodes.

(**Hint**: Assume that there exists a linear time algorithm SEARCH$(A, start, end, data)$ that searches for *data* in the array A between the indices *start* and *end* and returns the corresponding index. You may also find it useful to use a static variable.)

(a) (6 points) Two arrays in, pre corresponding to the inorder and preorder walks of a Binary Tree.

Solution:

```
1 TreeNode ReconstructBinaryTree(int[] preorder, int[] inorder)
2     prelen = preorder.length-1
3     inlen = inorder.length-1
4     TreeNode root = RSBTPIS(preorder,inorderTraversal,0,prelen,0,inlen)
5     return root
```

```
1 TreeNode RSBTPIS(int[] preorder, int []inorder, int prestart, int preend, int instart, int inend)
2     if (prestart>=preend||instart>=inend)
3           return null
4     TreeNode node = new TreeNode(preorder[prestart])
5     index = Seach(inorder,instart,inend,preorder[prestart])
6     node.left = RSBTPIS(preorder,inorder,prestart+1,prestart+index-instart,instart,index-1);
7     node.right= RSBTPIS(preorder,inorder,prestart+index-instart+1,preend,index+1,inend);
8     return node
```

In this algorithm we carefully examine that the first element of the preorder string is the root of that tree. And according to the rule of inorder traversal, the elements before that root node is the leftchild elements and the elements after that root node is the rightchild.

We just identify the root node and split the string to two parts which is left part and the right part. And recursively doing this process again and again. Then we get the whole tree.

We set the finishing condition and then get the result.

$$T(n) = T(k) + T(n-k-1) + O(n) = O(nlogn)$$

□

(b) (2 points) Two arrays post, pre corresponding to the preorder and postorder walks of a Binary Tree.

Solution:

It is not possible. □

(c) (4 points) One array pre corresponding to the preorder traversal of a ***Binary Search Tree***.

Solution:

```
1 TreeNode reconstructBST(int[] pre)
2     TreeNode root = reCBST(pre,0,pre.length-1)
3           return root
```

```
1 TreeNode reCBST(int[] pre, int preStart, int preEnd)
2     if (preStart>preEnd) return null;
3     TreeNode node = new TreeNode(pre[preStart]);
4     int index = 0;
5     for(int i=preStart;i<=preEnd;i++)
6           if (pre[i]<pre[preStart] and pre[i+1]>=pre[preStart])
7                 index = i-preStart;break;
8           else index = i-preStart;
9     node.left = reCBST(pre,preStart+1,index);
10    node.right = reCBST(pre,index+1,preEnd);
11    return node;
```

This is quite the same with what we do in the question(a). We have to find the boundary of leftchild and rightchild. According to the property of BST and preorder traversal, we find that the first element of this array is the root node.

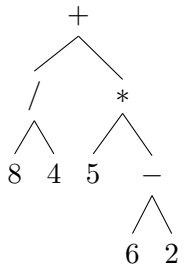
Then we keep looking for an element that it is smaller then the root node but its next is greater than or equal to the root node. Then this element is the boundary of its leftchild. Then we recursively use the function to split the left and right part of its children.

Finally We get the whole tree.

T(n) = T(k)+T(n-k-1)+O(n) = O(nlogn)

□

(d) (6 points) One array post corresponding to the postfix expression of an expression tree. The following expression: $(8/4) + (5 * (6 - 2))$ can be represented as the following tree:



Note that the inorder traversal of this expression tree yields the original expression and thus it is called the *infix* expression. The postorder traversal of this expression tree yields: $84/562 - *+$ and this is called the *postfix* expression. Postfix and Prefix expressions are often preferred to infix expressions because it avoids ambiguity about the order of operations. (**Hint**: Consider using a stack. Also, note that the operators are binary operators. You may also use boolean function isOperator which takes a character and returns True if it is an operator, else returns False. You may assume that the given expression is valid.)

Solution:

```
1 TreeNode reconExpTree(int[] post)
2     if (post ==null) return null;
3     LinkedList<TreeNode> stack = new LinkedList<TreeNode>();
5     for(int i=0;i<post.length;i++)
6           if (ISOperator(post[i]))
7                 If(stack.isEmpty() or stack.size()<2)
8                       return error;
9                 TreeNode node = new TreeNode(post[i]);
10                node.left = stack.pop();
11                node.right = stack.pop();
12                stack.push(node);
13          else
14                TreeNode node = new TreeNode(post[i]);
15                stack.push(node);
16    if (stack.isEmpty() or stack.size()>1)
17          return error;
18    return stack.pop();
```

□

This whole process is very similar to the process of caculating the value of postfix expressions. We construct a stack and check the post string one by one. If it is a number we just push it into the stack. If it is an operator we just pop out two elements to be its leftchild and rightchild. Then we push it back to the stack.

After we finish checking the whole string we will have only one node in the stack. And we just pop it out to be the root node.

If we can not pop out twice when looking for both two children, or after checking all the elements we have more than 1 node in this stack, then we get the wrong postfix expression.

The running time is

$$O(n)$$

CSCI-GA.1170-001/002 Fundamental Algorithms October 20, 2019

Solutions to Problem 3 of Homework 6 (10 (+5) points)

Name: jingshuai jiang (jj2903) *Due: 5 pm on Thursday, October 17*

Collaborators: NetID1, NetID2

Assume you are given an array of n integers with many duplicate values, so that you know that there are at most t distinct values in the array. The goal of this problem is to develop a sorting algorithm that runs in time $O(n \log t)$.

(a) (5 points) Build a data structure D which supports two operations: INSERT(D, v) and FREQUENCY(D, v). The functionality of FREQUENCY(D, v) is to return the number of times that INSERT(D, v) was called. Both operations should run in time $O(\log t)$ where t is the number of distinct values of v for which INSERT(D, v) has been called.

In other words if INSERT has been called 4 times on $(D, 2)$ and 2 times on $(D, 6)$ then FREQUENCY$(D, 3)$ returns 0 but FREQUENCY$(D, 2)$ returns 4 and both calls take time (about) $\log t$ where $t = 2$.

Solution:

In order to make have a $logt$time of insert and check the Frequency, we build a AVL tree which is an auto-balanced-BST. Recall that if the BST is extremely unbalanced,we will have a time of $o(n)$ to insert,delete and search. But if we have an auto-balanced-BST, we will have a stable insert time of $O(logt)$.

And we can get the Frequency of some specific value by adding an attribute to this Treenode data structure,each time we insert the same value,we add it by one. Then we just need to get the node's count numbers. All we have to do is to find that node and get its count attribute. The following is the datastructure of this AVL tree.

```
columns
  package com.company;
  import java.util.LinkedList;
  import java.util.List;
  public class AVLTree {
      private class AVLNode{
          Integer val;
          AVLNode left;
          AVLNode right;
          int height;
          int count;
          AVLNode(Integer value)
          {
              this(value,null,null);
          }
          AVLNode(Integer value, AVLNode leftchild,AVLNode rightchild)
          {
              val = value;
              left = leftchild;
              right = rightchild;
              height = 0;
              count=1;
          }
      }
      private  AVLNode root;
      public AVLTree()
      {
          root = null;
      }
      private int height( AVLNode node )
      {
          return node == null ? -1 : node.height;
      }
      public void insert(int value)
      {
          root = insert(value,root);
      }
      private AVLNode insert( Integer value, AVLNode node )
      {
          if( node == null )
              return new AVLNode( value, null, null );
          int compareResult = value.compareTo( node.val );
          if( compareResult < 0 )
          {
              node.left = insert( value, node.left );//insert the value into the lefttree
              if( height( node.left ) - height( node.right ) == 2 )//break the balance
                  if( value.compareTo( node.left.val ) < 0 )//LLtype
                      node = rotateWithLeftChild( node );
                  else   //LRtype
                      node = doubleWithLeftChild( node );
          }
          else if( compareResult > 0 )
          {
              node.right = insert( value, node.right );//insert the value into the righttree
              if( height( node.right ) - height( node.left ) == 2 )//break the balance
                  if( value.compareTo( node.right.val ) > 0 )//RR type
                      node = rotateWithRightChild( node );
                  else                              //RLtype
                      node = doubleWithRightChild( node );
          }
          else
          {
              node.count++;
          }
              ;  // repeat doing nothing
          node.height = Math.max( height( node.left ), height( node.right ) ) + 1;//update the height
          return node;
      }
      private AVLNode rotateWithLeftChild( AVLNode k2 )
      {
          AVLNode k1 = k2.left;
          k2.left = k1.right;
          k1.right = k2;
          k2.height = Math.max( height( k2.left ), height( k2.right ) ) + 1;
          k1.height = Math.max( height( k1.left ), k2.height ) + 1;
          return k1;
      }
      private AVLNode rotateWithRightChild( AVLNode k1 )
      {
          AVLNode k2 = k1.right;
          k1.right = k2.left;
          k2.left = k1;
          k1.height = Math.max( height( k1.left ), height( k1.right ) ) + 1;
          k2.height = Math.max( height( k2.right ), k1.height ) + 1;
          return k2;
      }
      private AVLNode doubleWithLeftChild( AVLNode k3 )
```

```
        {
            k3.left = rotateWithRightChild( k3.left );
            return rotateWithLeftChild( k3 );
        }
        private AVLNode doubleWithRightChild( AVLNode k1 )
        {
            k1.right = rotateWithLeftChild( k1.right );
            return rotateWithRightChild( k1 );
        }
        private Integer Frequency(Integer seachvalue,AVLNode node)
        {
            if(node ==null) return 0;
            int compareResult = seachvalue.compareTo( node.val );
            if( compareResult < 0 ) return Frequency(seachvalue,node.left);
            else if(compareResult >0) return Frequency(seachvalue,node.right);
            else return node.count;
        }
        public List<Integer> inorderTraversal()
        {
            return inorderTraversal(root);
        }
        private List<Integer> inorderTraversal(AVLNode root) {
            LinkedList<Integer> output = new LinkedList<>();
            LinkedList<AVLNode> stack = new LinkedList<AVLNode>();
            if (root == null) return output;
            AVLNode current = root;
            while (!stack.isEmpty() || current != null) {
                while (current != null) {
                    stack.push(current);
                    current = current.left;
                }
                current = stack.pop();
                while(current.count>0)
                {
                    output.add(current.val);
                    current.count--;
                }
                current = current.right;
            }
            return output;
        }
    }
```

□

(b) (5 points) Give an algorithm for sorting n integers with t distinct values in time $O(n \log t)$. You can use the data structure from part (a). Formally analyze the runtime of your algorithm and argue it's correctness.

Solution:

```
columns
    public List<Integer> AVLsort(int[] numbers)
    {
        AVLTree tree = new AVLTree();
        for(int i=0;i<numbers.length;i++)
        {
            tree.insert(numbers[i]);
        }
        return tree.inorderTraversal();
    }
```

The whole algorithm is quite simple. We use the insertion of the (a) part to insertion all the numbers into the AVL tree, each insertion takes time $O(logt)$, then all n numbers takes time$O(nlogt)$. And since avl is a balanced BST, then the inorderTraversal will get these numbers in an increasing order. Then we just need to do this inorderTraversal, which takes time $O(n)$,

Then the whole time takes about $O(nlogt + n) = O(nlogt)$ □

(c) (**Extra Credit**) (5 points) Generalize your algorithm from part (c) so that it operates on any comparison based set of objects rather then just integers.

Solution:

I just replaced the integer value to any class that implement Comparable function(others remain the same with D in (a)). Then this data structure and function can easily be applied to operates on any comparison based set of objects.

```
columns
    package com.company;
import java.util.LinkedList;
import java.util.List;
public class AVLComparableTree < T extends Comparable< ? super T>>{
    private class AVLCNode< T>{
        T element;
        AVLCNode< T> left;
        AVLCNode< T> right;
        int height;
        int count;
        AVLCNode< T>(Integer value)
        {
            this(value,null,null);
        }
        AVLCNode< T>(T thiselement, AVLCNode< T> leftchild,AVLCNode< T> rightchild)
        {
            element = thiselement;
            left = leftchild;
            right = rightchild;
            height = 0;
            count=1;
        }
    }
    private  AVLCNode< T> root;
    public AVLTree()
    {
        root = null;
    }
    private int height( AVLCNode< T> node )
    {
        return node == null ? -1 : node.height;
    }
    public void insert(T x)
    {
        root = insert(x,root);
    }
    private AVLCNode< T> insert( T x, AVLCNode< T> node )
    {
        if( node == null )
            return new AVLCNode< T>( x, null, null );
        int compareResult = x.compareTo( node.val );
        if( compareResult < 0 )
        {
            node.left = insert( x, node.left );//insert the value into the lefttree
            if( height( node.left ) - height( node.right ) == 2 )//break the balance
                if( x.compareTo( node.left.val ) < 0 )//LLtype
                    node = rotateWithLeftChild( node );
                else   //LRtype
                    node = doubleWithLeftChild( node );
        }
        else if( compareResult > 0 )
        {
            node.right = insert( x, node.right );//insert the value into the righttree
            if( height( node.right ) - height( node.left ) == 2 )//break the balance
                if( x.compareTo( node.right.val ) > 0 )//RR type
                    node = rotateWithRightChild( node );
                else                              //RLtype
                    node = doubleWithRightChild( node );
        }
        else
        {
            node.count++;
        }
        ;  // repeat doing nothing
        node.height = Math.max( height( node.left ), height( node.right ) ) + 1;//update the height
        return node;
    }
    private AVLCNode< T> rotateWithLeftChild( AVLCNode< T> k2 )
    {
        AVLCNode< T> k1 = k2.left;
        k2.left = k1.right;
        k1.right = k2;
        k2.height = Math.max( height( k2.left ), height( k2.right ) ) + 1;
        k1.height = Math.max( height( k1.left ), k2.height ) + 1;
        return k1;
    }
```

```
    private AVLCNode< T> rotateWithRightChild( AVLCNode< T> k1 )
    {
        AVLCNode< T> k2 = k1.right;
        k1.right = k2.left;
        k2.left = k1;
        k1.height = Math.max( height( k1.left ), height( k1.right ) ) + 1;
        k2.height = Math.max( height( k2.right ), k1.height ) + 1;
        return k2;
    }
    private AVLCNode< T> doubleWithLeftChild( AVLCNode< T> k3 )
    {
        k3.left = rotateWithRightChild( k3.left );
        return rotateWithLeftChild( k3 );
    }
    private AVLCNode< T> doubleWithRightChild( AVLCNode< T> k1 )
    {
        k1.right = rotateWithLeftChild( k1.right );
        return rotateWithRightChild( k1 );
    }
    private Integer Frequency(T seachelement,AVLCNode< T> node)
    {
        if(node ==null) return 0;
        int compareResult = seachelement.compareTo( node.val );
        if( compareResult < 0 ) return Frequency(seachelement,node.left);
        else if(compareResult >0) return Frequency(seachelement,node.right);
        else return node.count;
    }
    public List<T> inorderTraversal()
    {
        return inorderTraversal(root);
    }
    private List<T> inorderTraversal(AVLCNode< T> root) {
        LinkedList<T> output = new LinkedList<>();
        LinkedList<AVLCNode< T>> stack = new LinkedList<AVLCNode< T>>();
        if (root == null) return output;
        AVLCNode< T> current = root;
        while (!stack.isEmpty() || current != null) {
            while (current != null) {
                stack.push(current);
                current = current.left;
            }
            current = stack.pop();
            while(current.count>0)
            {
                output.add(current.val);
                current.count--;
            }
            current = current.right;
        }
        return output;
    }
}
```

□

CSCI-GA.1170-001/002 Fundamental Algorithms October 20, 2019

Solutions to Problem 4 of Homework 6 (10 points)

Name: jingshuai jiang (jj2903) *Due: 5 pm on Thursday, October 17*

Collaborators: NetID1, NetID2

(a) (5 points) You are given a binary tree. The task is to create a linear time algorithm PRUNE-SINGLE(*node*) that removes all internal nodes with one child, in the subtree rooted at *node*. Note that the leaf order should be preserved. Argue correctness and runtime of your algorithm.

Solution:

```
1 TREENODE PRUNE-SINGLE(Treenode node)
2     Prune-Single-subprogram(node,null,null)
3     return node
```

```
1 VOID PRUNE-SINGLE-SUBPROGRAM(node, parentnode, direction)
2     if (node ==NULL) return ;
3     if (node has only one child and parentnode ==NULL and direction==NULL)
4           return ;
5     if (node.leftchild==NULL and node.rightchild!=NULL)
6           parentnode.direction = node.rightchild
7           Prune-Single-subprogram(parentnode.direction,parentnode,direction);
8     if (node.rightchild==NULL and node.leftchild!=NULL)
9           parentnode.direction = node.leftchild
10          Prune-Single-subprogram(parentnode.direction,parentnode,direction);
11     else :
12          Prune-Single-subprogram(node.leftchild,node,leftchild);
13          Prune-Single-subprogram(node.rightchild,node,rightchild);
```

if the node is empty we just return null. If the rootnode has only one child, we will also leave it.

if the node only has its rightchild, we will get its original place replaced by this rightchild. And keep looking into this rightchild.If the node only has its leftchild, we will get its original place replaced by this leftchild. And keep looking into this leftchild.

If the node has its two children, we just keep looking into both of them.

Since in the worst case no node is has only one child then we will check all nodes one time. Then it is $O(n)$. □

(b) (5 points) You are given a *Binary Search Tree*. Create a linear time algorithm PRUNE-RANGE($node, start, end$) that removes all nodes in the subtree rooted at *node* whose key is not within the range $[start, end]$. Argue correctness and runtime of your algorithm.

Solution:

```
1 TREENODE PRUNE-RANGE(node, start, end)
2     Prune-Range-subprogram(node,start,end,null,null)
3     return node
```

```
1 VOID PRUNE-RANGE-SUBPROGRAM(node, start, end, parentnode, direction)
2     if (node ==NULL) return ;
3     if (node.value<start and parentnode==null)
4           Prune-Range-subprogram(node.right,start,end,node,right);
5           node.left =null
5     if (node.value>end and parentnode==null)
6           Prune-Range-subprogram(node.left,start,end,node,left);
7           node.right =null
8     if (node.value<start and parentnode!=null)
9           parentnode.direction = node.right
10          Prune-Range-subprogram(parentnode.direction,start,end,parentnode,direction);
11    if (node.value>end and parentnode!=null)
12          parentnode.direction = node.left
13          Prune-Range-subprogram(parentnode.direction,start,end,parentnode,direction);
14     else :
15          Prune-Range-subprogram(node.left,start,end,node,left);
16          Prune-Range-subprogram(node.right,start,end,node,right);
```

We construct this by looking at each node. if the rootnode itself's value is less than the start, then we prune its leftchild to be null and kept looking into its rightchild. if the rootnode itself's value is larger than the end, then we prune its rightchild to be null and kept looking into its leftchild.

Then we look into the subtree nodes if the node is null we jsut return. If the node's value isless than the start, then we prune both itself and its leftchild, and get its original place raplaced by its rightchild. If the node's value is larger than the end, then we prune both itself and its rightchild, and get its original place raplaced by its leftchild. Then we keep looking for its subtree.

Since in the worst case no node is out of the range then we will check all nodes one time. Then it is $O(n)$. □