CSE 100 Advanced Data Structures	Fall 2018
Homework 1	
Instructors: *Leo Porter* *(Sec. A)* & *Debashis Sahoo* *(Sec. B)*	**Due on:** Tuesday 16 October *(24 points)*

Name: ShihHan Chan PID: A15677346 *Date:* 10 / 11 / 2018

Instructions

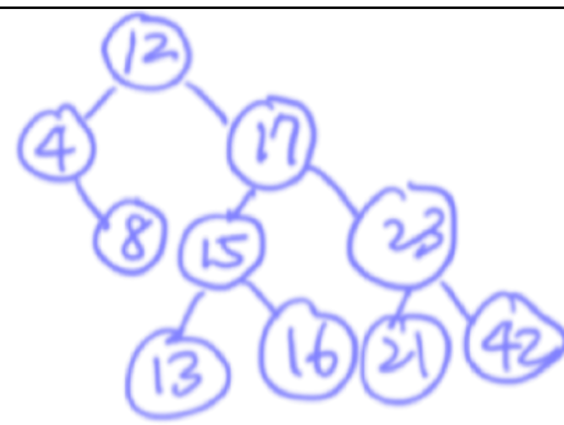
1. Answer each problem in the boxes provided. Any writing outside of the boxes will NOT be graded. Do not turn in responses recorded on separate sheets.
2. Handwritten or typed responses are accepted. In either case, make sure all answers are in the appropriate boxes.
3. All responses must be neat and legible. Illegible answers will result in zero points.

---

1. (4 points **- Correctness**) *BST Insertions* : Let T be the binary search tree created by inserting the following sequence of keys into an initially empty BST.

12, 4, 17, 8, 15, 16, 23, 42, 13, 21

(a) Draw the tree T.



(b) Find another sequence (different from the one above) that results in *exactly* the same tree as T.

12, 17, 4, 8, 15, 13, 16, 23, 21, 42

(c) *True or False* It is *always* possible to recover the original sequence that resulted in a binary tree. (Answer with **T for true**, or **F for false**.)

F

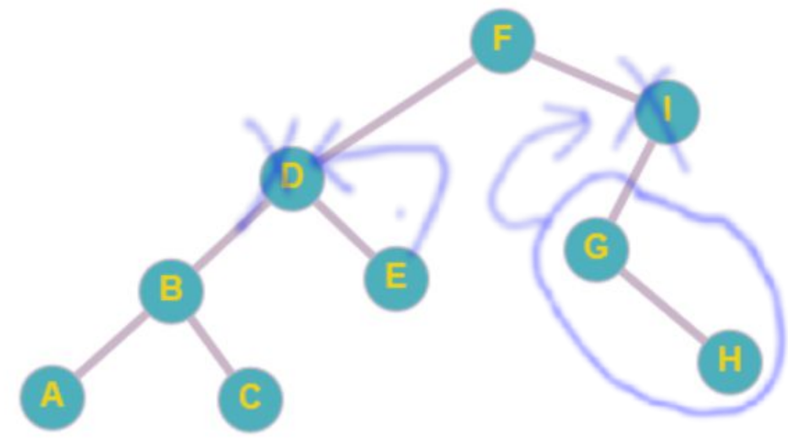
(d) What's the *expected (average)* number of comparisons to find an element in T, i.e. what is $E[find(x)] \mid x \in T$? We're *not* looking for a generic expression: your work will involve adding up fractions. Assume it takes one comparison to find an element if it's at the root, two comparisons if it's one level down, and so on. *Show and simplify your work.*

$\cancel{E[find(x)] \mid x \in T = \frac{1}{10} + \frac{2}{10} \times 2 + \frac{3}{10} \times 3 + \frac{4}{10} \times 4 = \frac{30}{10}}$

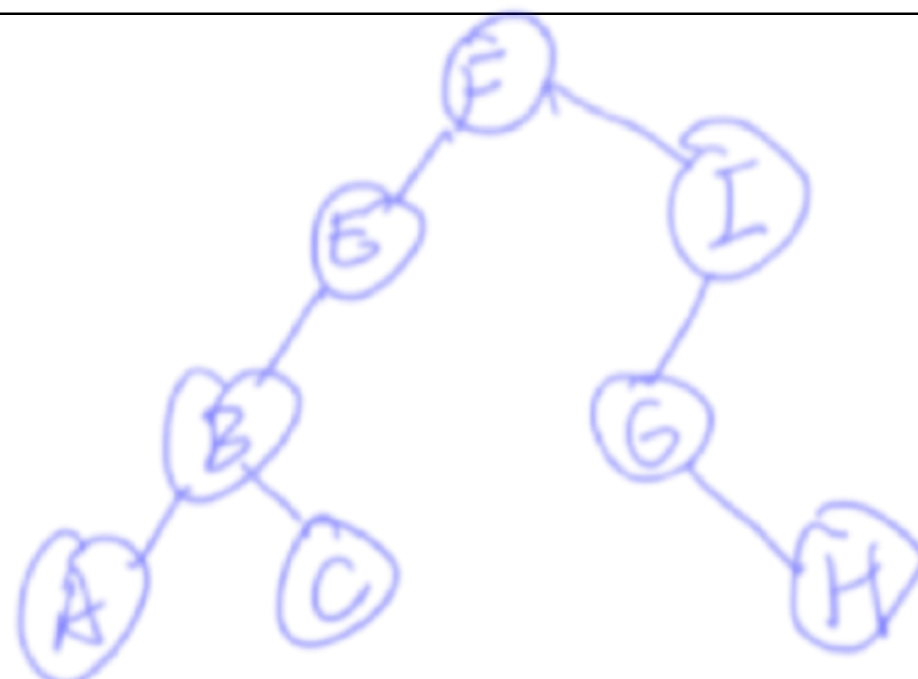
$\cancel{3}$

$E[find(x)] \mid x \in T = \frac{1}{10} + \frac{2}{10} \times 2 + \frac{3}{10} \times 3 + \frac{4}{10} \times 4 = 3$

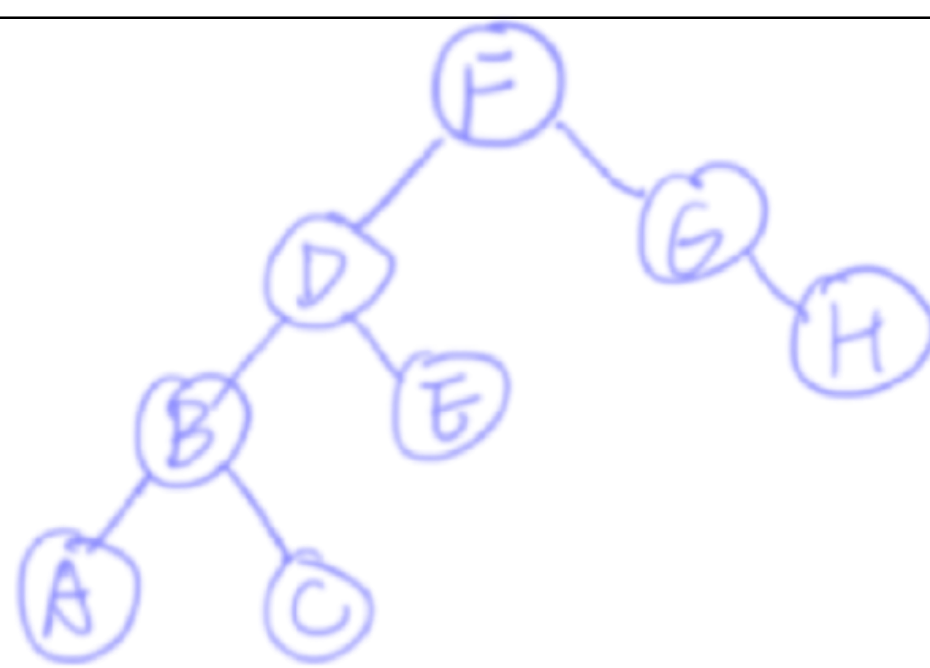
2. (4 points - **Correctness**) *BST Deletions*. Use the BST given below to answer each of the questions *independently*.



(a) Draw the above binary search tree after removing **D**.



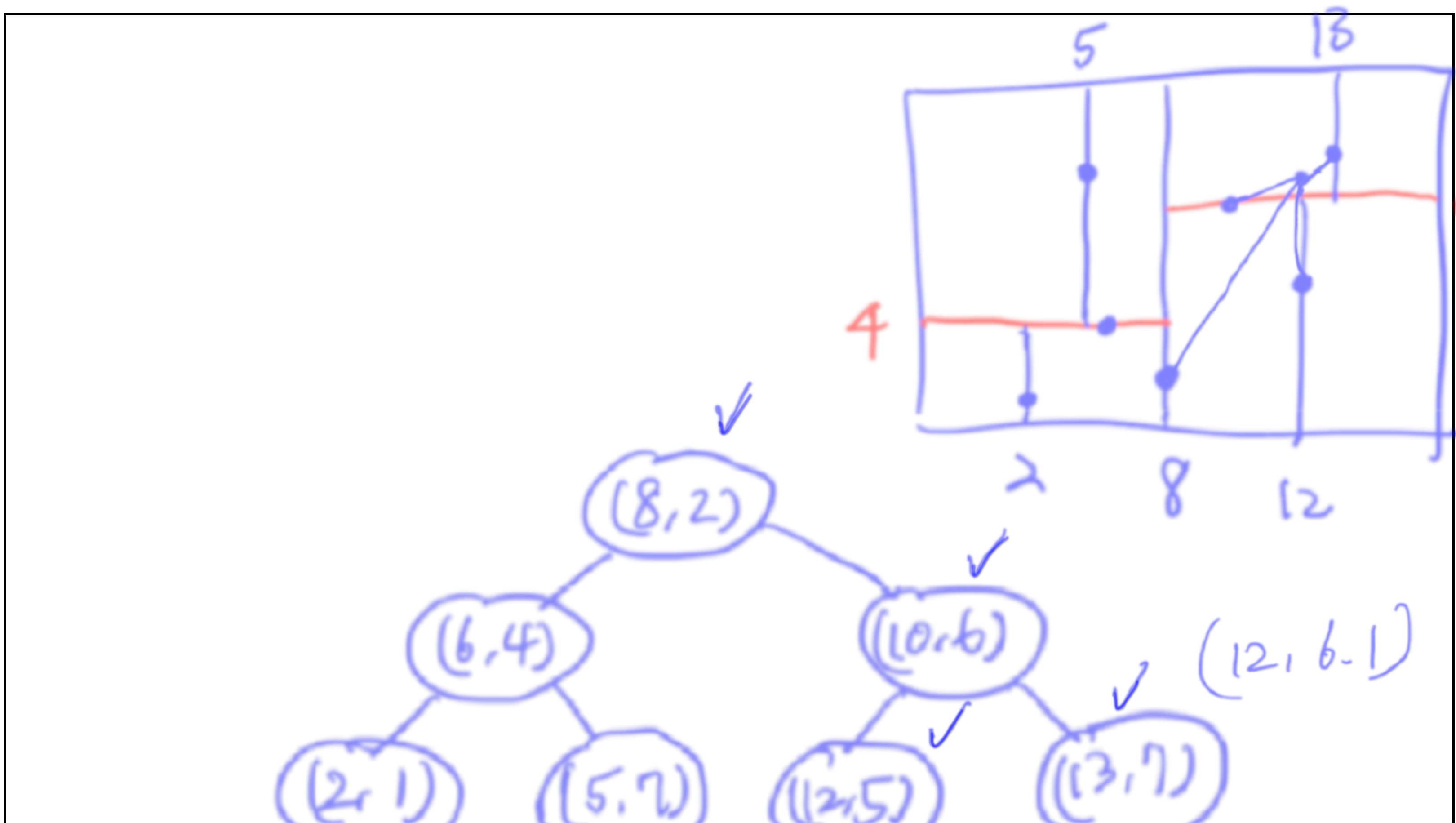
(b) Draw the above binary search tree after removing **I**.



3. (6 points - **Correctness**) *K-d Trees*. Let T be the 2-d tree built from the following sequence of points with *(x,y)* coordinates:

(8,2) ; (6,4) ; (5,7) ; (10,6) ; (13,7) ; (2,1) ; (12,5)

(a) Draw the tree T, using the typical visualization scheme for binary trees – i.e. *do not draw a K-d tree grid*. Make sure to include the coordinates of the nodes, as seen in lecture (i.e. root should be a circle with (8,2) inside it).



(b) Now, say we're querying for the nearest neighbor of point **(12, 6.1)**. Assume a *recursive* method `findNN` exists, with all the necessary arguments to recursively find the nearest neighbor.

How many nodes in the tree are evaluated as potential nearest neighbors of (12, 6.1)? Make sure to include the root node in your calculations. We only care about concrete nodes, recursive calls of `findNN` on `null` nodes don't matter.

4

e.g. `findNN((6,4),...)` looks at 2 nodes (the root node `(8,2)`, and `(6,4)`). `findNN((14,5),...)` returns `(12,5)` after evaluating 4 nodes as potential nearest neighbors: `(8,2)`, `(10,6)`, `(12,5)` and `(13,7)`.

(c) Finally, derive the *worst case* time complexity of the *build tree* operation shown in lecture (`BuildRecurse`), for building a K-d tree with n points known a priori. Your result should be a function of n and *must be tighter* than $O(n^3)$, but *does not* have to be the *tightest* upper bound. *Show and simplify your work.*

If we build the k-d tree with merge (quick) sort. The total complexity is $O(n\log(n)\log(n))$. Since there are $O(\log n)$ layers. (we split the tree by median), and sort function at each layer is $O(n\log n)$.

$2\times(\frac{n}{2}\log\frac{n}{2}) = n\log(\frac{n}{2}) = \theta(n\log n)$

4. (4 points - **Completeness**) *Lookup with BST*. You may have realized by now that BSTs are great for implementing lookup structures. As such, in this question, we wish to implement a Set using Binary Search Trees. You can assume you have a **`bst`** container with methods to support search, insert, delete and traverse. Given below are the headers to a C++ Set class and the `bst` container (implementation details omitted):

```
template <class T>
class Set {
private:
  // standard BST implementation
  bst<T> *tree;
public:
  Set() {
    tree = new bst<T>();
  }

  // insert will add an element to the BST
  // if the BST does not already contain it
  // returns: true if inserted,
  //          false if already in tree
  bool insert(const T& value);
};
```

```
template <class T>
class bst {
/* member variables, etc */
public:
  // returns true if found, false if not
  bool search(const T& value);

  // overwrites if duplicate
  void insert(const T& value);

  // true if deleted, false if not
  bool delete(const T& value);

  // values from in-order traversal
  vector<T> traverse();
};
```

Write the implementation of the **`insert`** method below.

```
bool Set::insert(const T& value){
        if(tree->search(value))return false;
        tree->insert(value);
          return true;}
```

5. (6 points - **Completeness**) *Invariant of a BST*. As seen in lecture, a binary search tree is a data structure in which all nodes have *at most two children*, and for which the following *invariant* must hold: the left descendents of any node are of *lesser* value than such node, whereas its right descendents are of *greater* value (duplicates not allowed in this case). Given the following `TreeNode` definition:

```
#include <vector>
using namespace std;

template <class T>
class TreeNode {
    public: /* for simplicity */
        T value;
        // left is children[0] and right is children[1]
        // but note there are no guarantees on the size of children...
        vector<TreeNode<T>*> children;
};
```

Implement the **`isBST`** function below. You may assume that `T` overloads all comparison operators, and feel free to define a helper method.

```
/* all necessary includes */
template <class T> // functions can also be templated how cool is that!
bool isBST(const TreeNode<T> * node)
    if(!node)return true;
    if(node->children[0]&&findbiggest(node->children[0]) >= node->value)return false;
    if(node->children[1]&&findsmallest(node->children[1]->value) <= node->value)return false;
    return isBST(node->children[0])&&isBST(node->children[1]);}

//make sure no null Treenode in this function's input(handle in isBST function)
template <class T>
T findsmallest(const TreeNode<T> * node){
   if((!node->children[0])&&(!node->children[1]))return node->value;
   if((node->children[0])&&(!node->children[1]))return min(node->value, findsmallest(node->children[0]));
   if((!node->children[0])&&(node->children[1]))return min(node->value, findsmallest(node->children[1]));
   return min(node->value,min(findsmallest(node->children[0]),findsmallest(node->children[1])));}

//make sure no null Treenode in this function's input(handle in isBST function)
template <class T>
T findbiggest(const TreeNode<T> * node){
   if((!node->children[0])&&(!node->children[1]))return node->value;
   if((node->children[0])&&(!node->children[1]))return max(node->value, findbiggest(node->children[0]));
   if((!node->children[0])&&(node->children[1]))return max(node->value, findbiggest(node->children[1]));
   return max(node->value,max(findbiggest(node->children[0]),findbiggest(node->children[1])));}
```