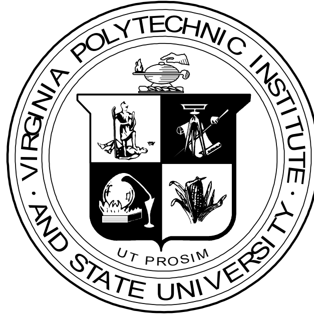


Instructions:

- Print your name in the space provided below.
- This examination is closed book and closed notes. No calculators or other computing devices may be used.
- Answer each question in the space provided. If you need to continue an answer onto the back of a page, clearly indicate that and label the continuation with the question number.
- If you want partial credit, justify your answers, even when justification is not explicitly required.
- There are 10 questions, priced as marked. The maximum score is 100.
- When you have completed the test, sign the pledge at the bottom of this page and turn in the test.
- **Note that either failing to return this test, or discussing its content with a student who has not taken it is a violation of the Honor Code.**

Do not start the test until instructed to do so!

Name Solution
printed

Pledge: On my honor, I have neither given nor received unauthorized aid on this examination.

____
signed

1. [10 points each] Given two non-negative functions f(n) and g(n), at most one of the following three statements is true:

   i) f is strictly O(g) (i.e., f is O(g) but not Θ(g)).
   ii) f is strictly Ω(g) (i.e., f is Ω(g) but not Θ(g)).
   iii) f is Θ(g).

   For each of the following questions, indicate which of the three possibilities above is true. Justify your conclusion rigorously, using any theorems covered in class that you like.

   a) $f(n) = 1000 - 7n + 2n^2$ and $g(n) = 5n^2$

   iii

   Each function is Θ of its dominant term, which is n^2 for each function. So, each of these functions is $\Theta(n^2)$ and so they are Θ of each other.

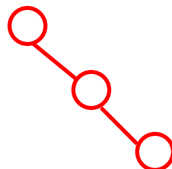
   b) $f(n) = 1000 - 7n + 2n^2$ and $g(n) = 6n^3$

   i

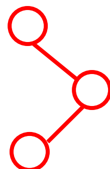
   f is $\Theta(n^2)$ and g is $\Theta(n^3)$. However, n^2 is strictly $O(n^3)$; this follows from the theorem that lists common complexity categories, or from taking the limit.

---

2. [5 points] Draw an AVL tree (just after a new node has been inserted) that would require a single rotation to re-balance it. The tree should contain as few nodes as possible and still meet the condition. Be sure to indicate the last node inserted. It is not necessary to show the data values or balance factors.



3. [5 points] Repeat the previous question, but with a double rotation required to re-balance the tree.



4. [10 points] Explain carefully but concisely what makes it possible to perform a binary search on an array of values, but impossible to perform a binary search on a linked list, assuming both structures store data in sorted order?

Because the elements of an array are stored contiguously, the address of each element can be calculated directly from the base address of the array and the index of the desired element. This also implies that the address of the midpoint of a segment of the array can be calculated directly, using simple arithmetic.

The nodes of a linked list are not stored contiguously, but rather are scattered randomly in the system heap. Averaging node addresses would yield no useful information.

---

5. [5 points] In the worst case, how many pointers that are already in the tree have to be reset in order to insert a new value into a BST (assume no parent pointers are used).

Insertion always adds a new leaf to the tree. The only pre-existing pointer that needs to be reset is the relevant one in the parent of the new leaf.

6. [5 points] In the worst case, how many pointers in tree nodes have to be reset to perform a single rotation in an AVL tree (assume no parent pointers are used).

In a single rotation, one pointer from the parent of the subtree root must be reset (unless the rotation is at the tree root, in which case the tree root pointer must be reset), as must the left child pointer of the left child pointer of the subtree root, and possibly the right child pointer of the subtree root.

For the next 3 questions, assume you have implementations of the BST and node template whose declarations follow:

```
template <typename T> class BinNodeT {
public:
   T             Element;
   BinNodeT<T>* Left;
   BinNodeT<T>* Right;

   BinNodeT();
   BinNodeT(const T& D, BinNodeT<T>* L  = NULL,
                        BinNodeT<T>* R = NULL);

   bool isLeaf() const;
   ~BinNodeT();
};

template <typename T> class BST {
private:
   BinNodeT<T>* Root;

   // private functions omitted. . .
public:
   BST();                                     // create empty tree
   BST(const BST<T>& Source);                 // deep copy support
   BST<T>& operator=(const BST<T>& Source);

   bool Insert(const T& D);               // insert node containing D
   bool Delete(const T& D);               // delete node containing D, if any
   T* const Find(const T& D);             // return pointer to D, if found
   void InOrderPrint(ostream& Out);       // display formatted tree
   void Clear();                          // delete all nodes, set Root to NULL
   ~BST();                                // delete all nodes
};
```

7. [12 points] Suppose a BST is created and N data values are inserted.

   a) In big-O terms, if the tree is well-balanced, what would be the average cost of inserting one more element?

   From the notes: log N.

   b) In big-O terms, assuming nothing about the structure of the tree, what would be the worst-case cost of inserting one more element?

   From the notes (if the tree is a stalk): N.

8. [8 points] What standard binary tree traversal would be used in deallocating all the nodes in the tree?

postorder: the parent node must not be deleted before its subtrees, in order to preserve the data needed to continue the traversal.

9. [15 points] We wish to add a public member function to the given BST template that will count and return the number of times a given data value occurs within the BST. The public function will look like:

```
template <typename T>
int BST<T>::Matches(const T& D) const {

   return MatchesHelper(D, Root);
}
```

where `MatchesHelper()` is a private, recursive member function. Complete the following implementation of the recursive helper function. Your implementation should not make any unnecessary visits to tree nodes.

```
template <typename T>
int BST<T>::MatchesHelper(const T& D, BinNodeT<T>* sRoot) const {

if ( sRoot == NULL ) return 0;   // no matches here, no data!

int matchesHere = 0;

if ( D == sRoot->Element )       // check for match at current node
   matchesHere = 1;

if ( D < sRoot->Element )        // only descend into relevant subtree

   return ( matchesHere + MatchesHelper(D, sRoot->Left) );

else

   return ( matchesHere + MatchesHelper(D, sRoot->Right) );

}
```

(If you need more room to show the implementation, you're not thinking about it correctly…. the following space is for drawing, if you find that helpful in analyzing the problem.)

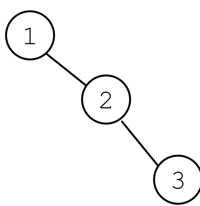
10. [15 points] We wish to add a public member operator to the given BST template that will compare two BST objects for equality. The public operator will look like:

```
template <typename T>
bool BST<T>::operator==(const BST<T>& RHS) const {

   return Equals(Root, RHS.Root);
}
```

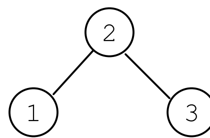
where `Equals()` is a private, recursive member function. Complete the following partial implementation of the recursive helper function. Your implementation should not make any unnecessary visits to tree nodes.

```
template <typename T>
bool BST<T>::Equals(const BinNodeT<T>* lRoot, const BinNodeT<T>* rRoot) const {

   if ( lRoot == NULL && rRoot == NULL ) return true;

   if ( lRoot == NULL || rRoot == NULL )   // if one is empty and one is not
      return false;

   if ( lRoot->Element != rRoot->Element ) // check for mismatch at current
      return false;

   // Here, both subtrees must be compared:

   return ( Equals( lRoot->Left , rRoot->Left ) &&

            Equals( lRoot->Right , rRoot->Right ) );
}
```

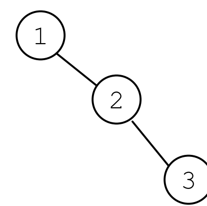
Note: two BSTs are equal if and only if they store the same values <u>and</u> have the same physical structure. Here, T_1 and T_2 are not equal, but T_1 and T_3 are:



T1



T2



T3