# **Exam 2**

*This is a closed book and closed notes test.* You are not allowed to have anything on your desk other than pencil and this exam paper during the test; this includes *calculators* or *electronic assistance* of any kind – ***especially smartphones***.

*You may not leave to go to the restroom.* Please go before the exam starts.

*You may not ask questions.* If something is confusing, write a note beside the question and explain your assumptions.

*You must show all of your work on this exam.* You will not be allowed to turn in additional sheets of paper.

*Read and sign the following statement.*  Failure to sign the statement will result in a **zero** on the exam.

*I have neither given nor received unauthorized assistance on this test. I have notified the proctor of any violations of the above policies.*

Signature: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

|  |  |
| --- | --- |
| **Problem** | **Score** |
| 1 | / 30 |
| 2 | / 20 |
| 3 | / 24 |
| 4 | / 26 |
| **Total** | **/ 100** |

*Points divided evenly among parts of a problem unless otherwise specified.*

1. [30 points] In the following AVL Trees, **a single node was just inserted but the tree has not yet been balanced**. In the first column, mark the node or nodes that could have been inserted. In the second column, indicate which nodes are unbalanced and their balance factors. In the third column, draw the balanced tree and indicate the rotations applied. If the tree is already balanced, simply write “no rotations” and do not re-draw the tree.

|  |  |  |
| --- | --- | --- |
| **AVL Tree**  **(mark inserted node(s))** | **Which nodes are imbalanced?** | **Draw the balanced tree and list rotations applied.** |
|  | [See scanned doc](https://drive.google.com/open?id=0B9UVdYe5Fn5WT0F3S3BCQ1BKamhNR1luODNnY294QmFBclZR) |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

2. [20 points] Using the following class definition for a node, implement the following. Use valid C++ syntax (including case for keywords) to earn full points.

class BSTNode {

public:

int data;

BSTNode \*left; // pointer to left sub-tree

BSTNode \*right;// pointer to right sub-tree

BSTNode(int d, BSTNode \*l=nullptr, BSTNode \*r=nullptr) {

data = d;

left = l;

right = r;

}

};

(a) [10 points] Insertion on a Binary Search Tree using the following function prototype. See comments below for more requirements.

// BST\_insert arguments:

// root is a reference to the root of the binary search tree. It

// could be null.

//

// newNode is newly allocated and initialized node that’s being

// inserted into the tree.

void BST\_insert(BSTNode\* &root, BSTNode\* newNode);

|  |
| --- |
| // Always redeclare function. This is a non-resursive version like  // I had in the notes.  void BST\_insert(BSTNode \* &root, BSTNode \*newNode)  {  if(root==nullptr) root = newNode;  else {  BSTNode \*tmp = root; //get copy so we don’t change root  while(tmp!=nullptr) // loop until we reach a leaf  // but we should break first  {  if (newNode->data < tmp->data) {  if(tmp->left == nullptr)  {  tmp->left = newNode;  break;  }  else tmp = tmp->left;  } else {  if(tmp->right == nullptr)  {  tmp->right = newNode;  break;  }  else  tmp = tmp->right;  } //inner if  } // while  } //else  }  //A simple recursive version is also possible becuse root is passed  //by reference. If it was not by reference, this would not work.  void BST\_insert(BSTNode \* &root, BSTNode \*newNode)  {  if(root==nullptr) root = newNode; //done  else // insert into tree below root  if (tmp->data < root->data) // go left?  // because root->left is passed as reference, we can  // change it in the recursive call if its nullptr  BST\_insert(root->left,newNode);  else // go right  // because root->right is passed as reference, we can  // change it in the recursive call if its nullptr  BST\_insert(root->right,newNode);  } |

(b) [10 points] Calculate the height of a Binary Search Tree using the following function prototype.

// BST\_height arguments:

// node is a pointer to the root of a sub-tree.

// Return the height of the sub-tree starting at node.

int BST\_height(BSTNode\* node);

|  |
| --- |
| // Redeclare function!  // Simplest approach is a recursive one  int BST\_height(BSTNode\* node)  {  // base case, stop on nullptr  **if (node==nullptr) return -1;** // could be 0, depends on next line  // not null, so this node must be 1 greater than max child.  // if no children at all, we should return 0. For no  // children, call will return -1 and we add 1, producing 0.  // of the two children, pick larger height from child and add 1  **return 1+std::max(BST\_height(node->left),BST\_height(node->right));**  } |

3. [24 points/ 3 points each] Provide short answers for the following questions about data structures.

1. In a max-heap, what is the index for the parent of the node with index 115?
2. In a max-heap, what are the indices for the children of the node with index 115?
3. What do we know about the value at the greatest valid index of a max-heap (assuming more than one value in the max-heap)?
4. What do we know about the value at the least valid index of a max-heap?
5. In a AVL tree of integers, what’s the Big-O time complexity of searching for a specific integer value in the tree?
6. In an AVL tree of integers, what do we know about the value at the root of the tree?
7. What’s the difference between a Binary Search Tree and an AVL tree?
8. Give an example of an adjacency list using a small graph (4 vertices).

4. Consider the design of a hash table for integers in which each entry in the hash table array points to a binary search tree. For example, new items inserted in the table are inserted into the BST at the calculated hash index. Also, the hash table array is never resized, despite how many elements are inserted into the hash table. For this data structure, answer the following questions.

1. [2 points] Is this hash table design more like linear probing or chaining? Justify your answer.

|  |
| --- |
| Chaining; Instead of inserting into a list at each index, we’re inserting into a BST. |

1. [10 points] Assuming a table of length 10 and a hash function of key % 10, what does the hash table look like after the following insertions:

33,5,15, 25,13,103,17,7,99

[See scanned doc](https://drive.google.com/open?id=0B9UVdYe5Fn5WT0F3S3BCQ1BKamhNR1luODNnY294QmFBclZR)

1. [7 points] Consider N to be the number of data items in the hash table and M to be the size of the table array. What is the big-O time complexity for insertion if N is much less than M assuming a perfect hash function? Justify your answer.

|  |
| --- |
| O(1). This is just the same case as a hash table with chaining. The BST at each entry doesn’t change the basic assumptions. So, the answer is the same. |

1. [7 points] Using the same definitions for N and M as in part (c) but hashing may not be perfect, what is the big-O time complexity for insertion if N is much greater than M? Justify your answer.

|  |
| --- |
| Note: this problem asks about big-O of **INSERTION not search.**  O(H) or O(log N). The worst here is that all N nodes are added to a single binary search tree. Insertion into a tree is O(H), since we may have to traverse to the deepest part of the tree.  For a balanced tree, this yields O(log N).  If you misread the problem and assumed a traditional chaining approach using a linked list instead, then insertion would be O(1). Inserting into a linked list can always be done in constant time, either at the head or tail. |

**C++ Keywords**

In common with C:

auto const double float int short struct unsigned  
break continue else for long signed switch void  
case default enum goto register sizeof typedef volatile  
char do extern if return static union while

Unique to C++:

asm dynamic\_cast namespace reinterpret\_cast try  
bool explicit new static\_cast typeid  
catch false operator template typename  
class friend private this using  
const\_cast inline public throw virtual  
delete mutable protected true wchar\_t

Reserved words:

and bitand compl not\_eq or\_eq xor\_eq  
and\_eq bitor not or xor

**ASCII Table**

