CS261: Midterm Exam 2

NAME: _____

INSTRUCTIONS

- This is a 45 minute, closed-book exam containing **THREE** problems
- Look at the back of the exam for the C code for Problems 1 and 3
- Cases of academic dishonesty will be filed to the Office of Student Conduct

Problems	Max points	Earned points
1.1	10	
1.2	30	
2	24	
3	36	
TOTAL	100	

Problem 1. Hash Table with Singly Linked Lists – 10 points + 38 points

Problem 1.1. Pseudo Code – 10 points

Write a pseudo code for adding a new element to a hash table with singly linked lists. The element consists of the key and value. The hash table keeps the record of a total number of elements.

Problem 1.2. C Implementation – 30 points

Write a function in C for adding a new element to a hash table with singly linked lists, `addHT()`. **You may use only your functions, built-in C functions, and the C code provided for Problem 1.** Please do not modify the names of data types, variables, and functions that we provided.

```
/* Add a data element to a Hash Table */
/* Input:
   ht -- pointer to a hash table with singly linked lists
   elem -- element to be added to the hash table consisting of the key and value.
*/


void addHT (struct HashTable * ht, struct DataElem elem) {

   /* FIX ME */
```

Problem 2. AVL Tree – 24 points

The main function of a C code, first, initializes an AVL tree, then, adds the sequence of numbers {3, 2, 8, 5, 9, 4} to the AVL tree, and, finally, prints all node values of the AVL tree in the pre-order fashion.

1) (12 points) Draw by hand the resulting AVL tree after adding all numbers from the sequence.
2) (12 points) Write the output of this main function.

C Code for Problem 1

```
#define TYPE_KEY char
#define TYPE_VALUE double
#define MAX_LOAD_FACTOR 10

struct HashTable {
  struct Link **table;/*Array of lists*/
  int count;/*number of elements*/
  int tablesize;/*number of lists*/
}

struct Link {
   struct DataElem elem;
   struct Link * next;
}
```

```
struct DataElem {
   TYPE_KEY key[100];
   TYPE_VALUE value;
}

/* The hash function */
int HASH(TYPE_KEY * key){
   int i;
   int index = 0;
   for (i = 0; key[i] != '\0'; i++)
      index += key[i];
   return index;
}
```

```
/* Double the size of a hash table with singly linked lists
      input: ht -- pointer to the hash table
      pre: the input hash table exists in the memory, and was initialized,
      post: the hash table size has doubled
*/
void _resizeHT(struct HashTable *ht) {
   int oldsize = ht->tablesize;
   struct HashTable *oldht = ht; /* old table */
   struct Link *cur, *last;
   int i;

   /* Allocate new memory with double the size, and initialize */
   ht->table = (struct Link **) malloc(sizeof(struct Link *) * (2 * oldsize));
   assert(ht->table != 0);
   ht->tablesize = 2 * oldsize;
   ht->count = 0;
   for(i = 0; i < ht->tablesize; i++)   ht->table[i] = 0;

   /* Copy the old hash table */
   for( i = 0; i < oldsize; i++) {
      cur= oldht->table[i];
      while(cur != 0){
         addHT(ht, cur->elem);
         /* remove the old link */
         last = cur;
         cur = cur->next;
         free(last);
      }
   }
   /* Free old table */
   free(oldht);
}
```

Problem 3: BST2AVL – 36 points

Write the C function, `bst2avl()`, for balancing the height of **all nodes** in the input binary search tree (BST). **You may use only your functions, built-in C functions, and the C code provided for Problem 3.** Please do not modify the names of data types, variables, and functions that we provided.

```
/* Transforms the input BST to the corresponding AVL tree */
/* Input: tree -- binary search tree
   Post: tree -- points to the corresponding AVL tree
*/
void bst2avl(struct Tree *tree){
  assert(tree);

  /* FIX ME */
```

C Code for Problem 3

```
# define TYPE int

struct Node {
  TYPE   val;
  struct Node *left;
  struct Node *right;
  int height;
};

struct Tree {
  struct Node *root;
  int size;
};

/* return height of current*/
int height(struct Node *current){
  if (current == 0) return -1;
  return current->height;
}

/* set height for current node */
void setHeight (struct Node * current){
  int lch = height(current->left);
  int rch = height(current->right);
  if (lch < rch)
    current->height = 1 + rch;
  else
    current->height = 1 + lch;
}

/* rotate right current node */
struct Node * rotateRight(struct Node * current){
  struct Node * new = current->left;
  current->left = new->right;
  new->right = current;
  setHeight(current);
  setHeight(new);
  return new;
}

/* rotate left current node */
struct Node * rotateLeft (struct Node * current){
  struct Node * newtop = current->right;
  current->right = newtop->left;
  newtop->left = current;
  setHeight(current);
  setHeight(newtop);
  return newtop;
}
```

HEIGHT GET

HEIGHT SET

ROT RIGHT

ROT LEFT

C Code for Problem 3 (continued)

BALNODE

```
/* Balance the height of the input node */
struct Node * balanceNode (struct Node * current){
   assert(current);
   /*  compute rotation flags */
   int rotation = height(current->right) - height(current->left);
   int dRotationRight = height(current->left->right) - height(current->left->left);
   int dRotationLeft = height(current->right->left) - height(current->right->right);

   if (bf < -1)
   {
     /* right rotation */
     /* check if double rotation is needed */
     if (dRotationRight > 0) {
        /* rotate left current's left child*/
        current->left = rotateLeft(current->left);
     }
     /* return the balanced node */
     return rotateRight(current);
   }
   else if(bf > 1)
   {
     /* left rotation */
     /* check if double rotation is needed  */
     if( dRotationLeft > 0 ){
        /* rotate right current's right child */
       current->right = rotateRight(current->right);
     }
     /* return the balanced node */
     return rotateLeft(current);
   }

   /* return current unchanged */
   return current;
}
```