Theory8

1.

(4)(a)

```
Solution:
bool is_ht(ht H) {
   if (H == NULL) return false;
   if (!(H->m > 0)) return false;
   if (!(H->n >= 0)) return false;
   //@assert H->m == \length(H->table);

   int nodecount = 0;

   for (int i = 0; i <_______H->m_______; i++)
   {
      //  set p equal to a pointer to first node
      //  of chain i in table, if any

      chain* p = ______H->table[i]______ ;

      while (__!p__)
      {
         elem e = p->data;

         if ((e == NULL) || (___elem_key(e)___!= i))

            return false;

         nodecount++;

         if (nodecount >___H->n___)

            return false;

         p =_____p->next______;

      }
   }

   if (_____nodecount != H->n_____)

      return false;

   return true;
}
```

(1)(b)

Solution:

```
/*@ensures \result ==__NULL_ || key_equal(k,___elem_key(\result)__);
 @*/
```

2.

(1)(a)

Solution:

```
          75
         /   \
       13      92
      /  \    /  \
    20   42  84  99
           \     /
           71   98
             \
             73
```

(1)(b)

Solution：42

(3)(c)

Solution:

```
int tree_height(tree* T)
//@requires is_ordered(T, NULL, NULL);
{
    if (T == NULL)
          return 0;

    int left = 1 + tree_height(T->left);
    int right = 1 + tree_height(T->right);
    if (left < right)
         return right;
    return left;

}

int bst_height(bst B)
//@requires is_bst(B);
//@ensures is_bst(B);
{
          return _______tree_height(B)________;
}
```

(5)(d)

Solution:

```
tree* tree_delete(tree* T, key k)
{
    if (T == NULL) {                        // key is not in the tree

        return ______NULL______;

    }

    if (key_compare(k, elem_key(T->data)) < 0) {

        __________T->left__________ = tree_delete(T->left, k);

        return T;

    } else if (key_compare(k, elem_key(T->data)) > 0) {

        __________T->right__________= tree_delete(T->right, k);

        return T;

    } else {// key is in current tree node T

        if (T->left == NULL)                // node has only right child

            return ________T->right________;

        else if (T->right == NULL)          // node has only left child

            return ________T->left________;

        else {                  // Node to be deleted has two children

            if (T->left->right == NULL) {

                //  Replace the data in T with the data
                //  in the left child.

                __________T->left->right = T->right__________;

                //  Replace the left child with its left child.

                __________T->right = NULL__________;

              return T;
            }
            else {
```

```
                //  Search for the largest child in the
                //  left subtree of T and replace the data
                //  in node T with this data after removing
                //  the largest child in the left subtree. T->data
                = largest_child(T->left); return T;

            }
        }
    }
}

elem largest_child(tree* T)
//@requires T != NULL && T->right != NULL;
{
    if (T->right->right == NULL) {

        elem e = ____________T->right->data______________;

        T->right = ____________T->right->left______________ ;

        return e;
    }

    return largest_child(________T->right->righ->data______);
}
```