

## 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)

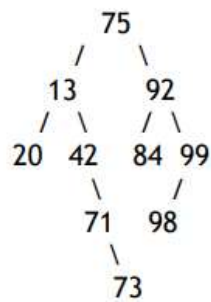
Solution:

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

2.

(1)(a)

Solution:



(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 _____;
    }

    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;
        }
    }
}
```

```

// 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->right->data _____);
}

```