## Hash Tables: Data Structure Invariants

（4a）扩展is\_ht，添加代码以检查哈希表中的每个元素是否与其所在的链匹配，并且每个链是否是非循环的。

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;

}

(1b) 考虑下面给出的ht\_lookup函数:

elem ht\_lookup(ht H, key k)

//@requires is\_ht(H);

{

int i = abs(hash(k) % H->m);

chain\* p = H->table[i];

while (p != NULL)

//@loop\_invariant is\_chain(p, i, H->m);

{

//@assert p->data != NULL;

if (key\_equal(elem\_key(p->data), k))

return p->data;

else

p = p->next;

}

/\* not in chain \*/

return NULL;

}

给出这个函数的一个简单的后置条件。

**作答如下：**

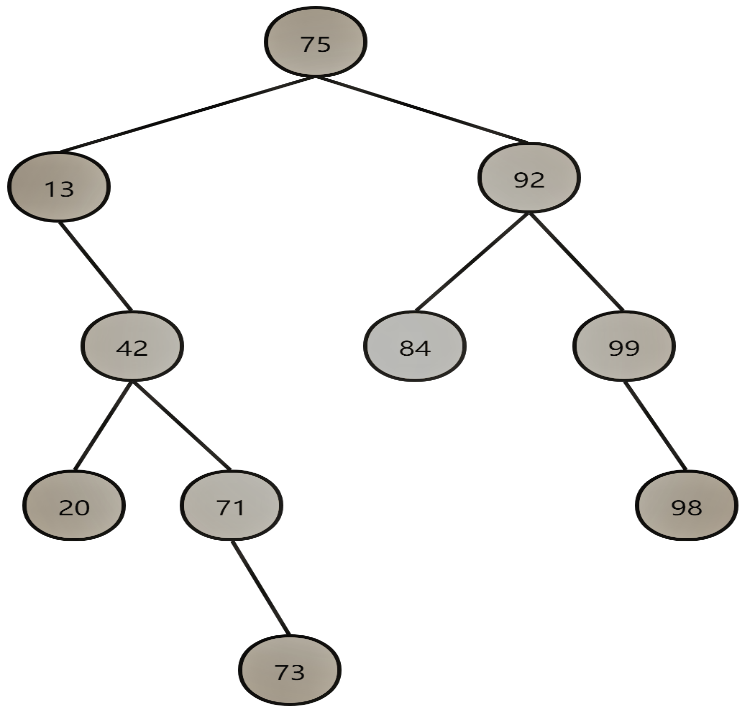
/\*@ensures \result == **NULL** || key\_equal(k,  **elem.key(\result)** );

@\*/

## Binary Search Trees

（1a）画出按以下给定顺序插入以下关键字得到的二叉查找树：

75 92 99 13 84 42 71 98 73 20



（1b） 如果以下五个关键字可以以任意的顺序插入，那么可以构造多少个不同的二进制搜索树：73、28、52、-9、104 ？

**（I）包含一个节点的二叉搜索树，只有一种情况；**

**（II）包含两个节点的二叉搜索树，总共有2种情况。**

**（III）包含3个节点的二叉搜索树，任选其中一个节点为根，只需计算剩下的两个节点所能构成的子树的种类；**

**（IV）仿此可得f(n) = f(n-1) + f(n-2)\*f(1) + … + f(1)\*f(n-2) + f(n-1);**

**（v）代入节点5，得f(5) = f(4) + f(3)\*f(1) + f(2)\*f(2) + f(1)\*f(3) + f(4) = 42**

**因此，总计可构造42种二叉搜索树**

（3c） 编写一个新的库函数bst\_height的实现，该实现返回二叉搜索树的高度。二叉搜索树的高度定义为你遵循从根到叶的路径时的最大节点数。因此，一个空的二叉搜索树的高度为0。

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);

//@requires is\_bst(B);

{

return tree\_height(B);

}

（5d）使用以下函数来扩展BST库实现，该函数使用给定的关键字从树中删除一个元素。

void bst\_delete(bst B, key k)

//@requires is\_bst(B);

//@ensures is\_bst(B);

{

B->root = tree\_delete(B->root, key k);

}

完成bst\_delete函数所使用的递归辅助函数tree\_delete的代码。一旦删除关键字（如果它在树中），此函数应该返回一个指向位于T的树的指针。

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** );

}