**Homework 9\_1 [Hashing.cpp]**

**Variable analysis**

#define KEY\_SIZE 10 : the maximum length of string.

#define TABLE\_SIZE 13 : the Bucket size in hash table.

ListNode\* hash\_table[TABLE\_SIZE] : hash table.

key : starting address of char array (=string)

|  |  |
| --- | --- |
| **element** | |
| **type** | **name** |
| char [KEY\_SIZE] | key |

Item : data used in hash key.

link : the address of connected node

|  |  |
| --- | --- |
| **ListNode** | |
| **type** | **name** |
| element | item |
| ListNode \* | link |

**Function analysis**

void hash\_chain\_delete(element item, ListNode\* ht[])

void hash\_chain\_delete(element item, ListNode\* ht[]) {

int hash\_value = hash\_function(item.key); //get hash value

ListNode\* node\_before = NULL;

ListNode\* node;

// find node with item.

for (node = ht[hash\_value]; node; node\_before = node, node = node->link) {

if (equal(node->item, item))

break;

}

// if node is NULL , there is no element corresponding to given item.

if (node == NULL) {

printf("delete key does not exist\n");

return;

}

ListNode\* temp = node;

if (node\_before)

{

node\_before->link = node->link;

printf("not null");

}

else //node\_before is null

{

ht[hash\_value] = node->link;

printf("null");

}

free(temp); //deallocate memory

}

: In order to delete certain key in the hash table, we not only know the node with key `itself`, but also the **before node** of the key. If node is null, which indicates fails to find given element, just return. The only thing left after finding what I mentioned is to connect before node to next node of key node. There are two cases in hash chain deletion.

1. Node\_before is NULL

: It means that the key node is the headmost of the bucket it is included in.

So substitude ht[hash\_vaule] for second head of the bucket ,which is completely same as node->link.

1. Node\_before is not NULL

: It means that the key node has both before and after node. So substitude node\_befor->link for next node of key node (=node->link)

**[result]**

텍스트이(가) 표시된 사진

자동 생성된 설명

**텍스트이(가) 표시된 사진

자동 생성된 설명**

**Homework 9\_2 [Bst\_sort.cpp]**

**Binary search tree**

****

**Binary Search Tree** is a node-based binary tree data structure which has the following properties:

* The left subtree of a node contains only nodes with keys lesser than the node’s key.
* The right subtree of a node contains only nodes with keys greater than the node’s key.
* The left and right subtree each must also be a binary search tree.

To use Binary Search Tree for sorting data, we have to apply proper tree traversal considering characteristics of BST. In BST left child <= vertex <= right is always true. So traversing with `Left -> vertex -> right` is needed, which is inorder traversal.

For example, if we apply inoder traversal to above graph, the visiting sequence is

1 -> 3->4->6->7->8->10->13->14. It is the desired result with acending order(sorted).

**Variable analysis**

data : the key used in bst insertion

left : the left child node

right : the right child node

|  |  |
| --- | --- |
| **Node** | |
| **type** | **name** |
| int | data |
| Node \* | left,right |

**function analysis**

1) int random(int data\_maxval)

// Integer random number generation function between 1 and n

int random(int data\_maxval) {

return rand() % data\_maxval + 1;

}

: Returns the generated random number not overring maximum value.

2) Node\* bst\_insert(Node\* root, int key)

테이블이(가) 표시된 사진

자동 생성된 설명

node\* bst\_insert(node\* root, int key) {

if (root == NULL) { // if find proper place to be inserted

// generate new node.

root = (node\*)malloc(sizeof(node));

root->data = key;

root->left = root->right = NULL;

}

else if (root->data >= key) // if key is the smaller

root->left = bst\_insert(root->left, key); // move to left tree

else // if key is the larger

root->right = bst\_insert(root->right, key); // move to right tree

return root;

}

: A new key is always inserted at the leaf. We start searching a key from the root until we hit a leaf node. Once a leaf node is found, the new node is added as a child of the leaf node.

3) void in\_order(Node\* root)

// l -> v - > r

void in\_order(node\* root) {

if (root == NULL)

return;

in\_order(root->left); // left

printf("%d\n", root->data); // vertex

in\_order(root->right); // right

}

: Using inorder(left->vertex->right)traversal, it prints out all nodes in the BST in acending order.

**[result]**

****